Linguaggio Macchina del commodore 64

EDIZIONE ITALIANA

David Lawrence
Mark England

I listati dei programmi riportati nel testo sono gli originali forniti dall'autore, e ad essi si riferiscono i valori di "checksum".
Su cassetta sono invece registrati gli stessi programmi con messaggi e commenti in italiano.

Linguaggio Macchina del Commodore 64

David Lawrence
Mark England

GRUPPO
EDITORIALE
JACKSON
Via Rosellini, 12
20124 Milano

SOMMARIO

PREFAZIONE

Questo non è un altro libro sul linguaggio macchina di un popolare microcomputer che consuma metà delle sue pagine spiegando tutte le istruzioni macchina del 6502/6510 ed i loro svariati modi di indirizzamento, con un bieco programma caricatore ed una serie di routine in linguaggio macchina presentate al termine e di dubbio interesse. L'intenzione di questo libro è quella di fornire ai lettori un solido programma in BASIC per immettere nel computer routine in linguaggio macchina ed in assembler e, contemporaneamente, offrire una serie di programmi in linguaggio macchina che vale la pena possedere.

Il programma BASIC si chiama Mastercode ed è uno strumento praticamente completo per la programmazione in linguaggio macchina, contenente un Monitor per permettervi di esaminare e cambiare il contenuto della memoria, un Disassemblatore che traduce i programmi in linguaggio macchina nella forma di un assembler, ed infine un File Editor ed un Assembler che permettono lo sviluppo di programmi in assembler e la loro trasformazione in codice macchina.

Per quanto riguarda le routine in linguaggio macchina della seconda metà del libro, abbiamo adottato un nuovo approccio. Troverete una raccolta di routine che potrete usare per ampliare il BASIC del vostro 64 con 14 nuovi comandi. Oltre ad accrescere la potenza del BASIC, vi aiuterà ad imparare le tecniche della vera programmazione in linguaggio macchina ed il modo in cui il programmatore in linguaggio macchina può sfruttare le routine presenti nell'interprete BASIC del 64.

Il libro non vuole essere un introduzione al linguaggio macchina. Ciò non significa che non possa essere usato da principianti. Semplicemente abbiamo dato per scontato che voi possediate, o possiate accedere, a qualche altro testo che spieghi in dettaglio ogni istruzione del 6502/6510. Riteniamo che, concentrandoci sui programmi e spiegando le tecniche usate in essi, abbiamo potuto offrire un lavoro di maggior valore. Se vi capiterà di controllare qualche strana istruzione sul vostro libro-base del 6502, quello sarà il piccolo prezzo che dovrete pagare.

Tutti i programmi del libro sono stati controllati, Mastercode in particolare è stato provato fino all'esaurimento (nostro). Le routines in linguaggio macchina sono state tutte sviluppate su Mastercode, poichè questo era l'unico modo per essere sicuri che voi sareste riusciti a fare lo stesso. Se vi imbatterete in errori di qualsiasi

tipo la colpa sarà nostra, ma osiamo dire che non dovrebbero essere gravi, vista l'attenzione usate nel testing dei programmi.

Il libro è un lavoro svolto in cooperazione fra due persone di provenienza assai differente. David Lawrence scrive principalmente programmi BASIC ed è autore di parecchi libri sui microcomputer. Il suo interesse per l'hardware è minimo. Mark England studia Ingegneria Elettronica, si trastulla con i chips di silicio come se fossero la sua vera natura ed ha imparato a scrivere in linguaggio macchina quando si è bruciata la ROM contenente il BASIC del suo micro. Ha scoperto che non ne aveva più bisogno.

L'idea venne a David Lawrence, stufo di comperare libri sul linguaggio macchina che non gli fornivano nulla di veramente interessante quando vi lavorava sopra seriamente. Mark England è autore della maggior parte della programmazione, sebbene non senza imbarazzo. La forma finale del libro deriva della necessità di Mark England di spiegare al suo co-autore di cosa si occupassero i suoi programmi e di farli tradurre per i comuni mortali. In questo processo di spiegazione e discussione si è sviluppato uno stile che, crediamo, renda giustizia sia ai programmi sia alla necessità dei lettori di comprenderli.

Altro partner nella stesura del libro è stato il Commodore 64. Questo testo non sarebbe stato possibile per molti altri micro. È solo grazie alla filosofia Commodore di aprire la macchina al programmatore, lasciando libero accesso ad una moltitudine di routines dell'interprete che altri si sforzano di mettere sotto chiave, che siamo riusciti a scomporre e ricomporre il BASIC del 64. Abbiamo avuto discussioni con il 64, ma esso resta una macchina inestimabile per chi voglia andare oltre il BASIC.

Infine, dobbiamo ringraziare il personale dei negozi Microchip di Winchester e Southampton, che ci ha fornito aiuto insostituibile al momento di sostituire attrezzature e rifornimenti. Dobbiamo ringraziamenti anche a Jane Lawrence, che regolarmente, andando a letto mentre noi picchiavamo sulla tastiera a notte inoltrata, ci diceva quale meraviglioso lavoro stessimo facendo.

Speriamo che avesse ragione.

Parte I

CAPITOLO 1
MONITOR MASTERCODE

Ogni programma, non importa in quale linguaggio sia scritto, inizia la propria vita come una serie di istruzioni codificate ed immagazzinate nella memoria di un computer. Nel caso della maggior parte dei linguaggi, le istruzioni che compongono il programma sono totalmente prive di senso per l'unità centrale di elaborazione o CPU, il computer-nel-computer che viene alla fine chiamato ad eseguire i compiti imposti dal programma.

Per superare questo problema, a metà fra il programma immesso dall'utente e la CPU si troverà un altro programma, molto spesso inserito nella macchina al momento della fabbricazione, che si occupa di tradurre il programma dell'utente in una forma che la CPU sia in grado di comprendere.

Il programma 'inserito' permanentemente, comunque, svolge un'altra funzione, dal momento che senza il suo aiuto sarebbe in primo luogo impossibile per l'utente inserire le istruzioni.

Dal momento in cui il computer viene acceso, il programma incorporato inizia il suo compito esaminando la tastiera per rilevare un input dal mondo esterno. In seguito raccoglie questi input e li memorizza in modo tale che possano poi venir interpretati dalla CPU. L'utente che scrive programmi in BASIC raramente è conscio di questo processo. Le linee di programma vengono scritte, il tasto RETURN premuto e le linee diventano parte del programma – a patto che si osservi la grammatica BASIC correttamente.

Non è richiesto alcuno sforzo o pensiero particolare per inserire una nuova istruzione, al termine o nel corpo del programma, dal momento che la memoria del computer viene automaticamente ridisposta per far posto al nuovo ingresso.

Quando si passa a programmare in linguaggio macchina la situazione non è così semplice. Non ci sono facilitazioni incorporate nel computer che permettano di immettere nuove istruzioni dalla tastiera con la sicurezza che saranno automaticamente memorizzate ed il contenuto della memoria riorganizzato per far loro posto. Il primo compito di un programmatore in linguaggio macchina è perciò quello di realizzare un metodo per introdurre le istruzioni, esaminare la memoria e riorganizzarla per soddisfare le necessità crescenti del programma che si sta sviluppando. Questo vale se le istruzioni macchina sono inserite direttamente sotto forma di nu-

meri (forma in cui alla fine essi si presentano alla CPU) oppure per mezzo di uno speciale linguaggio detto 'linguaggio assemblatore' che facilita l'immissione e la comprensione dei programmi in linguaggio macchina. Lo strumento più semplice che permette di realizzare questa gestione della memoria si chiama 'Monitor', ed in questo capitolo costruiremo un programma Monitor flessibile che permetterà di esaminare singoli bytes di memoria o grossi pezzi di essa e di modificarne a piacere il contenuto.

SEZIONE 1: Inizializzazione e Menu

MODULO 1.1

```
10000 REM********************************
10020 REM GENERAL INITIALISATION
10030 REM********************************
10031 BASE = 16
10032 IF LEN(PTR$)+LEN(E$)<>255 THEN CLR : GO
SUB 19000
10035 DEV = 1
10040 DEFFN HEX(X) = (X AND 15)+48-((X AND 15
)>9)*7
10050 DEFFN DEC(X) = X-48+(X>57)*7
10060 FALSE = 0 : TRUE = -1
10070 POKE 53281,1 : POKE 53280,15
```

Lo scopo di questo modulo è quello di inizializzare un certo numero di variabili che verranno usate più avanti nel programma. La funzione di queste variabili è spiegata brevemente nella tabella presentata in Appendice, ma una piena comprensione sarà possibile solo quando le sezioni seguenti del programma, in cui le variabili vengono usate, verranno presentate. A questo punto è sufficiente semplicemente immettere correttamente il modulo — l'unico effetto visibile della sua esecuzione sarà il cambio di colore dello schermo.

TAVOLA DI CONTROLLO

```
10000 123     10020 238     10030 123
10031 116     10032 164     10035 2
10040 182     10050 132     10060 21
10070 220
```

Questa tavola di controllo è qui per aiutarvi a controllare che non abbiate fatto errori nell'introdurre un programma lungo e complesso. Per capire come usarla, si veda l'appendice A, che contiene il programma di controllo da aggiungere al termine del programma Mastercode quando iniziate, mettendovi in grado di generare le vostre tabelle di controllo da confrontare con quelle del testo.

MODULO 1.2

```
19000 REM********************************
19001 REM TEMPORARY LINES
19002 REM********************************
19010 RETURN
19011 REM ***END OF MONITOR PROGRAM***
```

Questo modulo temporaneo è posto a questo punto nel programma per permettere la chiamata, da parte della routine di inizializzazione, di una sezione del programma Mastercode che verrà inserita in seguito. Le linee di questo modulo verranno in futuro cancellate con l'immissione delle prossime sezioni del programma Mastercode. Il formato non è problematico, così non c'è bisogno di tabelle di controllo.

MODULO 1.3

```
10100 REM********************************
10101 REM CONTROL ROUTINE FOR MONITOR
10102 REM********************************
10110 DATA EXIT TO BASIC,MEMORY MODIFY,MEMORY
 DUMP,MACHINE CODE EXCUTE
10111 DATA LOAD MACHINE CODE FILE,SAVE MACHIN
E CODE FILE
10120 DATA DISASSEMBLER
10130 DATA FILE EDITOR
10140 DATA ASSEMBLER
10190 DATA END
10200 RESTORE
10220 X = 0
10230 PRINT "[BLU][CLH]--------- MACHINE CODE
 MONITOR ---------[GRN][CDW]"
10250 READ T$
10260 IF T$<>"END" THEN PRINT TAB(5) X ">" T$
 : X= X+1 : GOTO 10250
```

```
10265 IF X<15 THEN FOR Y = X TO 15 : PRINT :
NEXT
10270 PRINT "COMMAND ( 0 -" X-1 " ) : ";: INP
UT T
10300 IF T<0 OR T>X THEN 10100
10305 IF T=0 THEN PRINT "[CLH][CDW][CDW][CDW]
[CDW][RON]BYE[ROF][CDW][CDW][CDW][CDW]" : CLO
SE 1 : END
10310 ON T GOSUB 13100,13300,13500,14300,1410
0,15800,24800,20000
10320 GOTO 10100
```

Ogni programma complesso deve fornire all'utente un mezzo per scegliere quale delle molte funzioni dovrà essere usata in seguito. Questa possibilità è realizzata grazie ad una routine di controllo o, più semplicemente, un menu. Il menu qui presentato è più complesso di quanto fosse necessario per il programma corrente. Questo è dovuto al fatto che il Monitor è progettato in modo da poter essere più avanti esteso con le sezioni successive dell'assemblatore Mastercode. Piuttosto che dover inserire nuove linee di programma per tener conto delle funzioni extra che verranno fornite, il menu si estenderà automaticamente per prendere nota dei nuovi nomi delle opzioni immesse nelle frasi DATA.

TAVOLA DI CONTROLLO

```
10100 123       10101 101       10102 123
10110 180       10111 62        10120 33
10130 170       10140 65        10190 122
10200 140       10220 122       10230 192
10250 31        10260 210       10265 160
10270 234       10300 8         10305 201
10310 199       10320 155
```

SEZIONE 2: Uscita su schermo del contenuto della memoria

In questa sezione del programma esamineremo quelle parti di programma necessarie per permetterci di visualizzare ordinatamente il contenuto di un'area di memoria specificata. I moduli commentati qui possono sembrare insignificanti e potreste chiedervi perché non sono stati uniti per formare un solo modulo. Conoscendo meglio il programma Mastercode vedrete che singoli moduli possono essere richia-

mati da parti differenti del programma. Limitare i moduli ad una ed una sola funzione particolare ci permetterà di risparmiare un certo numero di linee di programma evitando di doverle duplicare in altre sezioni di questo.

MODULO 1.4

```
11000 REM********************************
11001 REM CONVERT DECIMAL TO HEX
11002 REM********************************
11010 T = H : H$ = ""
11020 H$ = CHR$ (FNHEX(T-INT(T/16)*16))+H$ :
T = INT(T/16) : IF T>0 THEN 11020
11050 RETURN
```

Questo modulo di tre linee trasforma un numero decimale in un esadecimale, cioè in un numero a base 16 invece che 10. I programmatori in linguaggio macchina usano quasi universalmente i numeri esadecimali, per la semplice ragione che essi si conformano maggiormente al sistema aritmetico binario usato dal computer.

Il sistema di numerazione esadecimale ha le seguenti 16 cifre: 0 1 2 3 4 5 6 7 8 9 A B C D E F. I computer più moderni memorizzano i numeri in gruppi di 256 (da 0 a 255), e la ragione per cui l'esadecimale è più conveniente è che il massimo valore esprimibile con due cifre è 255 (15∗ per la cifra più significativa e 15 per l'altra). Usare gli esadecimali significa poter realizzare una rappresentazione molto più ordinata dei valori memorizzati. Inoltre, il sistema binario usato dal computer implica che molto spesso numeri esadecimali significativi come 1000, (o 4096 in decimale), lo siano anche in termini di operazioni del computer. Iniziare a pensare in esadecimale è un importante aiuto alla comprensione del funzionamento del micro.

Commento

11020: l'operato di questa linea è più comprensibile attraverso un esempio. Supponete che con la variabile H si sia memorizzato il numero decimale 4875. Per convertirlo in esadecimale, dobbiamo prima riconoscere che è composto da 1∗16↑3 (16↑3=4096) + 3∗16↑2 (16↑2=256) + 0∗16↑1 (16↑1=16) + 11∗16↑0 (16↑0=1).

Questa linea isola ogni unità di queste differenti potenze di 16 e le traduce quindi in un carattere che rappresenta la cifra esadecimale appropriata, usando la funzione definita dall'utente FNHEX (vedi linea 10040) per selezionare il carattere cor-

retto. Nel caso di 4875, il numero esadecimale corrispondente sarà 130B. Per le cifre con un valore compreso fra 0 e 9, FNHEX restituisce semplicemente il valore dell'appropriato carattere 0-9 (codici carattere: 48-57). Se il valore della cifra è fra 10 e 15, allora si aggiunge 7 al codice carattere per portarlo nell'ambito A-F dell'insieme di caratteri del 64.

TABELLA DI CONTROLLO

```
11000 123      11001 167      11002 123
11010 170      11020 74       11050 142
```

MODULO 1.5

```
11950 REM*******************************
11951 REM CONVERT HEX IN H$ TO DEC IN H
11952 REM*******************************
11975 ERR = FALSE : H = 0 : IF LEN(H$)=0 THEN
 12030
11980 FOR X = 1 TO LEN(H$)
11990 T = FNDEC(ASC (MID$ (H$,X,1))): H = H*B
ASE+T
12010 IF T>BASE-1 OR T<0 THEN ERR = TRUE
12020 NEXT X
12030 RETURN
```

Sebbene sia più comodo introdurre numeri esadecimali, l'uso del BASIC implica che essi debbano venir trasformati in numeri decimali per gli usi del programma. Questo è realizzato dal presente modulo.

Commento

11975: in tutto il programma si farà uso della variabile ERR (errore) per indicare che è stato scoperto un errore. Il valore normale di ERR sarà 0, che è il valore assegnato alla variabile FALSE nella routine di inizializzazione. Ogni volta che un errore viene scoperto, ERR è posta al valore TRUE, che è –1. Il vantaggio dell'uso di questi valori di 'verità' è che ERR può essere utilizzata in frasi come ER-R=(A >50). L'espressione fra parentesi ha un valore a seconda che sia vera o falsa. Se è falsa avrà valore 0, se vera –1.

In questo modo la variabile ERR può essere usata per mostrare che qualcosa non va in modo molto più economico che non usando frasi del tipo: IF A >50 THEN ERR=–1.

11980-12020: esaminando successivamente ogni carattere della stringa H$, , questo ciclo estrae il valore decimale del carattere esadecimale usando la funzione definita dall'utente FNDEC (linea 10050). Dal momento che il ciclo parte da sinistra, il risultato corrente va moltiplicato per 16 per ogni successiva cifra esadecimale. Se viene introdotto un carattere al di fuori dell'ambito 0-F, la variabile ERR viene posta a -1 per segnalarlo ai moduli successivi.

TAVOLA DI CONTROLLO

```
11950 123     11951 230     11952 123
11975 142     11980 128     11990 40
12010 208     12020 250     12030 142
```

MODULO 1.6

```
12050 REM*********************************
12051 REM INPUT START ADDRESS
12052 REM*********************************
12057 H$ = ""
12060 INPUT "START ADDRESS ( IN HEX ) : "; H$
 : GOSUB11950
12080 IF ERR OR H<0 OR H>65535 THEN 12060
12090 AD = H : RETURN
```

Per visualizzare sullo schermo il contenuto di un'area di memoria è necessario specificare la locazione di partenza in memoria. Questo viene fatto in esadecimale, l'input viene poi tradotto in decimale dal modulo precedente.

TAVOLA DI CONTROLLO

```
12050 123     12051 19      12052 123
12057 162     12060 50      12080 128
12090 199
```

MODULO 1.7

```
11850 REM*********************************
11851 REM ASK CONTINUE ?
11852 REM*********************************
```

```
11858 T$ = ""
11860 INPUT "CONTINUE ( Y/N ) : "; T$
11870 IF T$="Y" THEN CO = TRUE : GOTO 11895
11880 IF T$<>"N" THEN PRINT "[CUP]"; : GOTO 1
1850
11890 CO = FALSE
11895 RETURN
```

Questo modulo viene richiamato quando una zona di memoria è passata su video per chiedere all'utente se desidera proseguire con un'altra.

TAVOLA DI CONTROLLO

```
11850 123     11851 114     11852 123
11858 174     11860 34      11870 72
11880 235     11890 239     11895 142
```

MODULO 1.8

```
11100 REM********************************
11101 REM BYTE INTO HEX
11102 REM********************************
11110 H = PEEK(AD) : AD = AD+1
11120 GOSUB 11000
11130 IF LEN(H$)<2 THEN H$ = "0"+H$
11140 O2$ = O2$+H$
11150 RETURN
```

Questo modulo esegue concretamente il compito di estrarre un valore dalla locazione di memoria specificata dalla variabile AD. Quindi viene chiamato il modulo 2 per trasformare il valore in esadecimale. Le cifre esadecimali composte da un unico simbolo vengono completate con uno zero in prima posizione, in modo da assicurare un formato standard di due cifre per ogni byte della memoria. Infine il numero esadecimale viene sommato a 02$, per mostrare sullo schermo il contenuto della memoria.

TAVOLA DI CONTROLLO

```
11100 123      11101 66       11102 123
11110 35       11120 159      11130 223
11140 82       11150 142
```

MODULO 1.9

```
13300 REM********************************
13301 REM DUMP MEMORY TO SCREEN
13302 REM********************************
13310 GOSUB 12050
13320 PRINT "[CLH]" : FOR X1 = 1 TO 18 : H =
AD : GOSUB 11000
13340 O2$ = "" : O1$ = H$ : O3$ = ""
13350 FOR X2 = 0 TO 7
13360 GOSUB 11100 : O2$ = O2$+" "
13375 IF H>31 AND H<95 THEN O3$ = O3$+CHR$ (H
) : GOTO 13380
13377 O3$ = O3$+"."
13380 NEXT X2
13390 PRINT O1$ TAB(5) O2$ TAB(31) O3$
13400 NEXT X1
13410 PRINT : GOSUB 11850 : IF CO THEN 13320
13440 RETURN
```

Abbiamo ora presentato tutti i moduli necessari per definire un indirizzo iniziale e per prelevare dati dalla memoria. Possiamo ora procedere con la parte di programma che effettivamente fa qualcosa. Avendo definito un punto di inizio, questo modulo visualizza sullo schermo il contenuto di un'area di memoria.

Commento

13320: il ciclo X1 sarà usato per stampare 18 linee, ciascuna contenente otto valori letti in memoria a partire dall'indirizzo ora memorizzato in AD.

13350-13380: i valori esadecimali forniti dal modulo precedente vengono memorizzati nella stringa 02$. Se il valore contenuto in una particolare locazione di memoria è il codice ASCII di una lettera o di una cifra, quel carattere verrà immagazzinato nella stringa 03$, per venir visualizzato a fianco dei valori relativi ad esso.

Nellá massima parte dei casi i caratteri visualizzati non avranno senso, poiché risulta perfettamente casuale il fatto che un numero rappresenti il codice di un carattere stampabile. In ogni caso, quando si esaminano aree di memoria come l'area delle variabili del 64 oppure la struttura del programma BASIC stesso, o un programma macchina contenente stringhe, questa caratteristica risulterà indispensabile per dare un quadro esatto di ciò che la memoria contiene, dal momento che verranno visualizzate tutte le stringhe in essa presenti.

TAVOLA DI CONTROLLO

```
13300 123      13301 129      13302 123
13310 165      13320 246      13340 173
13350 104      13360 100      13375 177
13377 90       13380 44       13390 89
13400 43       13410 86       13440 142
```

Riassunto

Una volta introdotta nel computer questa sezione, avrete la base funzionante dell'intero programma. Nelle prossime sezioni, troverete impiegati molti dei moduli già descritti, infatti funzioni come la conversione in esadecimale sono comuni a tutte loro. Prima di procedere col resto del programma, familiarizzatevi con quanto visto finora. Esaminate ad esempio l'area di memoria che contiene il programma stesso (e che inizia alla locazione 801 esadecimale) e l'area delle variabili. Questa sezione del monitor è di per sè un potente strumento per scoprire i segreti della memoria del 64.

MONITOR: Visualizzazione della memoria dall'indirizzo 801 esad.

```
801   25 08 10 27 8F 2A 2A 2A   %..´.***
809   2A 2A 2A 2A 2A 2A 2A 2A   ********
811   2A 2A 2A 2A 2A 2A 2A 2A   ********
819   2A 2A 2A 2A 2A 2A 2A 2A   ********
821   2A 2A 2A 00 42 08 24 27   ***.B.$´
829   8F 20 47 45 4E 45 52 41   . GENERA
831   4C 20 49 4E 49 54 49 41   L INITIA
```

```
839  4C 49 53 41 54 49 4F 4E   LISATION
841  00 66 08 2E 27 8F 2A 2A   ....'.**
849  2A 2A 2A 2A 2A 2A 2A 2A   ********
851  2A 2A 2A 2A 2A 2A 2A 2A   ********
859  2A 2A 2A 2A 2A 2A 2A 2A   ********
861  2A 2A 2A 2A 00 74 08 2F   ****.../
869  27 42 41 53 45 20 B2 20   'BASE .
871  31 36 00 9B 08 30 27 8B   16...0'.
879  20 C3 28 50 54 52 24 29    .(PTR$)
881  AA C3 28 45 24 29 B3 B1   ..(E$)..
889  32 35 35 20 A7 20 9C 20   255 . .

CONTINUE ( Y/N ) :
```

SEZIONE 3: Modifiche della memoria

Avendo imparato come esaminare la memoria, procediamo ora al passo successivo, cioè ad alterarne il contenuto. In questa sezione presentiamo altri due moduli che vi permetteranno di muovervi avanti e indietro nella memoria, visualizzando il contenuto dei singoli byte e, se lo desiderate, di alterare il contenuto del singolo byte visualizzato.

MODULO 1.10

```
13000 REM*********************************
13001 REM GET 1 BYTE
13002 REM*********************************
13007 H$ = ""
13010 INPUT "BYTE ( IN HEX ) : "; H$
13030 GOSUB 11950
13040 IF ERR OR H<0 OR H>255 THEN PRINT
"[CUP]" : GOTO 13000
13050 RETURN
```

Sulla base dei moduli precedenti dovrete avere poche difficoltà nell'accertarvi che questo modulo accetta un valore esadecimale compreso fra 0-FF (0-225), attiva una traduzione in decimale e restituisce questo valore al modulo successivo, da cui è richiamato.

```
13000 123     13001 52      13002 123
13007 162     13010 171     13030 173
13040 192     13050 142
```

MODULO 1.11

```
13100 REM******************************
13101 REM MEMORY MODIFY
13102 REM******************************
13110 GOSUB 12050
13120 H = AD : GOSUB 11000 : PRINT H$ TA
B(6) "/" ; : O2$ = ""
13140 GOSUB 11100 : AD = AD-1 : PRINT H$
 SPC(6) ;
13150 T$ = ""
13160 INPUT " +,-,I,E : "; T$
13170 IF T$="+" AND AD<65535 THEN AD = A
D+1 : GOTO 13120
13180 IF T$="-" AND AD>0 THEN AD = AD-1
: GOTO 13120
13190 IF T$="E" THEN RETURN
13200 IF T$<>"I" THEN PRINT"[CUP][CUP]"
: GOTO13120
13210 GOSUB 13000 : POKE AD,H : GOTO 131
20
```

Lo scopo di questo modulo è quello di permettere all'utente di muoversi in memoria da un indirizzo di partenza specificato e modificare il contenuto di singoli byte. Gran parte del modulo si occupa di far visualizzare sullo schermo in modo comprensibile i valori di ciascun byte e di muoversi in memoria. I cambiamenti del contenuto della memoria sono eseguiti dall'ultima linea, che richiama anche il modulo precedente.

Commento

13120-13140: ottenuto l'indirizzo di partenza, viene visualizzato l'indirizzo del byte corrente, insieme al valore contenuto.

13160-13210: il modulo fa uso di quattro simboli particolari, o 'prompt': '+' significa passaggio al prossimo byte, '–' significa passaggio al byte precedente ed 'E' significa uscita da questo modulo. Il prompt rimanente è 'I', che serve a richiamare il modulo precedente e permette di inserire un nuovo valore nel byte corrente.

TAVOLA DI CONTROLLO

```
13100 123     13101 112     13102 123
13110 165     13120 220     13140 17
13150 174     13160 192     13170 75
13180 116     13190 211     13200 244
13210 229
```

MODULO 1.12

```
13500 REM********************************
13501 REM MACHINE CODE EXECUTE
13502 REM********************************
13510 GOSUB 12050 : SYS AD : RETURN
```

Qualora desideriate usare il monitor per introdurre programmi direttamente in memoria in forma esadecimale, questa routine di una linea vi permetterà di chiamare la routine di linguaggio macchina senza uscire dal programma. È cosa saggia non far eseguire programmi in linguaggio macchina prima di aver salvato il programma introdotto fino a quel momento.

TAVOLA DI CONTROLLO

```
13500 123     13501 18      13502 123
13510 106
```

SEZIONE 4: Salvataggio e caricamento del file

Ora che siete in grado di immettere nuovi valori in memoria e quindi di sviluppare un programma in linguaggio macchina, dovete anche avere la possibilità di salvare quei programmi che in seguito svilupperete ed immetterete. Dovete inoltre a-

vere la possibilità di richiamare quei programmi da disco o nastro, a seconda del mezzo con cui li registrerete. Le quattro brevi routine seguenti hanno il compito di offrirvi queste possibilità.

MODULO 1.13

```
11250 REM********************************
11251 REM INPUT FILE NAME
11252 GOSUB 25500 : IF DEV=4 THEN 11290
11255 IN$ = ""
11260 INPUT " FILE NAME : "; IN$ : T = L
EN(IN$)
11280 IF T>16 OR T<0 THEN PRINT "[CDW]FI
LE NAME INVALID" : GOTO 11260
11290 RETURN
```

Il salvataggio di un gruppo di informazioni su nastro o disco avviene raggruppandole in forma di 'file', cioè una locazione dotata di nome che deve essere anzitutto 'aperta' prima che le informazioni vi vengano immesse e quindi chiusa quando tutte le informazioni necessarie sono state memorizzate.

Quando le informazioni vengono richiamate dev'essere specificato il nome del file. Questo modulo permette di inserire il nome di un file.

TAVOLA DI CONTROLLO

```
11250 123     11251 192     11252 119
11255 241     11260 137     11280 216
11290 142
```

MODULO 1.14

```
11200 REM********************************
11201 REM INPUT FINISH ADDRESS
11202 REM********************************
11205 H$ = ""
```

```
11210 INPUT "FINISH ADDRESS ( IN HEX) :
"; H$ : GOSUB 11950
11230 IF ERR OR H<0 OR H>65535 THEN 1120
0
11240 EA = H : RETURN
```

I programmi linguaggio macchina che in seguito svilupperete con l'aiuto dei programmi di questo libro saranno poi contenuti in blocchi di memoria. Per salvarli il programma necessita di due informazioni: il punto di partenza ed il termine del blocco.

Abbiamo già una routine che ottiene l'indirizzo iniziale, questa esegue la stessa funzione per l'indirizzo finale.

TAVOLA DI CONTROLLO

```
11200 123     11201 70      11202 123
11205 162     11210 101     11230 123
11240 200
```

MODULO 1.15

```
14100 REM********************************
14101 REM MACHINE CODE SAVE
14102 REM********************************
14110 GOSUB 11250 : GOSUB 12050 : GOSUB
11200
14115 T$ = "N" : IF DEV=8 THEN INPUT "OV
ERWRITE EXISTING FILE ( Y/N ) : "; T$
14116 IF T$="Y" THEN IN$ = "@0:"+IN$
14120 IF DEV=8 THEN IN$ = IN$+",S,W"
14125 IF SA>EA THEN 14190
14130 OPEN 2,DEV,2,IN$ : PRINT# 2,AD : P
RINT# 2,EA
14150 FOR X = AD TO EA : PRINT# 2,PEEK(X
) : NEXT : PRINT# 2 : CLOSE 2
14190 RETURN
```

Ora che siamo in grado di dare un nome al file in cui sarà contenuta l'informazione presente in un'area di memoria e di specificare il punto di inizio e di termine, possiamo procedere ed immettere questo modulo, che memorizzerà l'informazione su nastro o disco.

Commento

14125: questa linea controlla semplicemente che l'utente non abbia definito un blocco di memoria la cui locazione finale sia precedente rispetto a quella iniziale.

14130: viene aperto un file, in questo caso un file d'uscita, e la destinazione dell'informazione viene stabilita in base al valore della variabile DEV (device = dispositivo). Nel listing di questo programma essa è posta uguale ad 1 (linea 10035), cosa che dirige l'uscita verso il registratore a cassette. Se usate un disco, DEV dovrebbe essere posta uguale ad 8 nella linea 10035. Una volta aperto il file d'uscita, le prime due informazioni da inserirvi sono l'indirizzo iniziale (AD) e finale (EA). Più avanti nel programma verrà aggiunta la possibilità di cambiare a piacere il numero di dispositivi.

14150: il contenuto di ogni byte del blocco di memoria da salvare viene memorizzato singolarmente nel file.

TAVOLA DI CONTROLLO

```
14100 123     14101 46      14102 123
14110 224     14115 32      14116 89
14120 179     14125 92      14130 108
14150 161     14190 142
```

MODULO 1.16

```
14300 REM*********************************
14301 REM MACHINE CODE LOAD
14302 REM*********************************
14310 GOSUB 11250 : IF DEV=8 THEN IN$ =
IN$+",S,R"
14320 OPEN 2,DEV,0,IN$ : INPUT# 2,SA,EA
: IF ST THEN CLOSE 2 : RETURN
14350 FOR X = SA TO EA : INPUT# 2,T : PO
KE X,T : NEXT : CLOSE 2 : RETURN
```

Questo modulo è semplicemente l'immagine speculare del precedente. Invece di mettere informazioni in un file, questo modulo preleva dal file informazioni precedentemente memorizzate e le reinserisce nella memoria del computer.

TAVOLA DI CONTROLLO

```
14300 123      14301 31      14302 123
14310 206      14320 174     14350 50
```

Sommario

Una volta immesso l'intero monitor, siete in grado di sfruttarlo, sebbene tutta la sua potenzialità verrà ben compresa quando immetterete il resto del programma Mastercode. Provate ad immettere una nuova linea: 0A=13. Chiamate l'opzione del menu che permette di cambiare la memoria ed alterate il contenuto del byte 805 immettendovi l'esadecimale 8F (143). Listate il programma e vedrete che la vostra prima linea è cambiata in una frase di commento (143 rappresenta REM nel file di programma). Se non siete ben sicuri di ciò che state facendo è utile che non cerchiate ora di modificare troppe altre locazioni di memoria e certamente non prima che abbiate salvato correttamente la vostra versione finale del monitor. Se volete fare un po' di confusione provate a modificare alcuni dei byte attributo dei colori nelle posizioni D800-DBFF esadecimali, della memoria colore dello schermo. Errori commessi in questa zona non saranno disastrosi.

CAPITOLO 2
DISASSEMBLER MASTERCODE

Dopo aver inserito il programma monitor, che vi permette di esaminare aree di memoria e di cambiarne il contenuto, proseguiamo col passo successivo, che vi permetterà di tradurre il contenuto di un'area di memoria in cui si trovi un programma in linguaggio macchina in una forma più comprensibile. Questa forma più 'leggibile' per un programma in codice macchina si chiama linguaggio assemblatore.

Il vantaggio di lavorare con il linguaggio assemblatore è che usando i POKE per immettere numeri direttamente in memoria possiamo immettere un programma in linguaggio macchina, ma non c'é una corrispondenza immediatamente evidente fra i numeri inseriti o riletti dalla memoria e le operazioni che il programma eseguirà. Il programma risulta semplicemente una lista di numeri e solo un numero molto ristretto di programmatori è in grado di leggere un programma scritto in quella forma senza usare una tabella di riferimento contenente i codici ed il loro significato. Il linguaggio assemblatore fornisce un mezzo per immettere istruzioni significative anche per l'utente con poca pratica, che danno una rappresentazione esatta di ogni istruzione macchina del programma. In altre parole il linguaggio assemblatore consiste di una serie di istruzioni, o codici mnemonici, corrispondenti alle singole operazioni macchina che il chip 6502/6510 è in grado di riconoscere ed eseguire.

Le istruzioni in linguaggio assemblatore saranno normalmente composte da due parti:

1) un codice operativo (opcode) che specifica il tipo di operazione che si chiede al chip 6502/6510 di compiere, come spostare un numero da un luogo ad un altro in memoria, confrontare due valori o eseguire un'operazione aritmetica su di un valore;
2) una volta definito il tipo di operazione da eseguire è necessario definire il numero su cui l'operazione dev'essere compiuta. Questa parte dell'istruzione è nota come operando e può consistere di un numero su cui si agirà direttamente o dell'indirizzo di memoria di un numero su cui si dovrà operare.

Una tipica istruzione in linguaggio macchina, su cui si basa la traduzione del linguaggio assemblatore, consisterà perciò di un byte che specifichi l'opcode e di uno o due byte usati per ritrovare il numero su cui eseguire l'operazione. Alcuni tipi di istruzione necessitano di un solo byte, quello che precisa il solo opcode, dal momento che esse implicano che il valore su cui operare sia rappresentato da una locazione prefissata che non è necessario menzionare esplicitamente.

Per tradurre un programma dal codice macchina in linguaggio assemblatore c'è bisogno di un programma che sia in grado di identificare un opcode e quindi di decidere quanti dei successivi byte di memoria (0,1 o 2) rappresentino l'operando associato a quel codice operativo. Un programma in grado di far questo è detto 'disassembler'. Il suo effetto è quello di estrarre numeri incomprensibili dalla memoria e di tradurli in qualcosa che con un po' di pratica possa essere compreso dall'utente.

La spiegazione breve e molto semplificata data sopra richiederà un po' di attenzione se questa è la prima volta che sentite parlare di un disassembler. Una volta però che abbiate ben chiaro il concetto nella vostra mente dovreste avere pochi problemi nel comprendere le basi su cui lavora la seguente sezione del programma Mastercode. Per mezzo di una serie di tabelle memorizzate in stringhe, il programma è in grado di identificare le istruzioni in linguaggio macchina di una specifica area di memoria e di visualizzare il tipo di operazioni ed i loro operandi in linguaggio assemblatore. Il programma può essere usato in almeno due modi:

a) per l'utente che sta sviluppando programmi in linguaggio assemblatore, il disassembler permette di controllare e correggere più facilmente il programma in memoria durante il procedimento di immissione;

b) per coloro che vorrebbero approfondire la loro esplorazione della memoria del Commodore 64, il programma così com'è è in grado di dare una traduzione completa della ROM della macchina, cioè del programma incorporato e permanente che fa in pratica funzionare la macchina. In questo modo si può giungere ad una comprensione migliore del lavoro interno del computer ed è possibile esaminare il modo in cui le singole routine della ROM possano venir usate effettivamente nei programmi utente.

SEZIONE I: Predisposizione delle tabelle

MODULO 2.1

```
12200 REM********************************
12201 REM HEX LOADER
12202 REM********************************
12210 T1$ = ""
12220 FOR X1 = 1 TO LEN(T$) STEP 2
12230 T1$ = T1$+CHR$ (FNDEC(ASC (MID$ (T
$,X1,1)))*16+FNDEC(ASC (MID$ (T$,X1+1,1)
)))
12260 NEXT X1
12270 RETURN
```

Il vero scopo di questo modulo non sarà chiaro finchè non verranno spiegate le tabelle del modulo successivo. La sua funzione è quella di estrarre i valori dalle tabelle e di compattarli in stringhe. I valori delle tabelle sono stati disposti nella forma di valori esadecimali a due cifre (cioè numeri nell'intervallo 0-255 decimale). Questo modulo converte coppie di valori esadecimali in singoli caratteri ASCII. I caratteri ottenuti in questo modo possono essere raggruppati in una stringa (TI$).

TAVOLA DI CONTROLLO

```
12200 123     12201 107     12202 123
12210 223     12220 216     12230 154
12260 43      12270 142
```

MODULO 2.2

```
19000 REM********************************
19001 REM INITIALISE DECODER TABLES
19002 REM********************************
19005 BASE = 16
19007 DEFFN DEC(X) = X-48+(X>57)*7
19010 DIM TA$(4)
19011 T$ =     "0A2238383822023824220238З
8220238"
19012 T$ = T$+"09223838382202380D2238383
8220238"
```

```
19013 T$ = T$+"1C01383806012738260127380
6012738"
19014 T$ = T$+"0701383838012738 2C0138383
8012738"
19015 T$ = T$+"2917383838172038231720381
B172038"
19016 T$ = T$+"0B17383838172038 0F1738383
8172038"
19017 T$ = T$+"2A00383838002838250028381
B002838"
19018 GOSUB 12200 : TA$(0) = T1$
19019 T$ =    "0C0038383800283 82E0038383
8002838"
19020 T$ = T$+"382F3838312F3038163835383
12F3038"
19021 T$ = T$+"032F3838312F3038372F36383
82F3838"
19022 T$ = T$+"1F1D1E381F1D1E38331D32381
F1D1E38"
19023 T$ = T$+"041D38381F1D1E38101D34381
F1D1E38"
19024 T$ = T$+"1311383813111438 1A1115381
3111438"
19025 T$ = T$+"0811383838111438 0E1138383
8111438"
19026 GOSUB 12200 : TA$(0) = TA$(0)+T1$
19027 T$ =    "122B3838122B1838192B21381
22B1838"
19028 T$ = T$+"052B3838382B18382D2B38383
82B18"
19029 GOSUB 12200 : TA$(0) = TA$(0)+T1$
19030 T$ =    "171116611201 1CC1381114411
B111AA1"
19031 T$ = T$+"C711666112 01CCC1381114411
B111AA1"
19032 T$ = T$+"17111661120 1CCC1381114411
B111AA1"
19033 T$ = T$+"17111661120 19CC1381114411
B111AA1"
19034 T$ = T$+"1711666111 11CCC1381144511
B111A11"
19035 T$ = T$+"2721666112 11CCC1381144511
B11AAB1"
19036 T$ = T$+"2711666112 11CCC1381114411
B111AA1"
19037 T$ = T$+"2711666112 11CCC1381114411
B111A"
```

```
19038 GOSUB 12200 : TA$(1) = T1$+CHR$ (1
60)
19040 TA$(2) =          "ADCANDASLBCCBCSBE
QBITBMIBNEBPLBRKBVCBVS"
19041 TA$(2) = TA$(2)+"CLCCLDCLICLVCMPCP
XCPYDECDEXDEYEORINCINX"
19042 TA$(2) = TA$(2)+"INYJMPJSRLDALDXLD
YLSRNOPORAPHAPHPPLAPLP"
19043 TA$(2) = TA$(2)+"ROLRORRTIRTSSBCSE
CSEDSEISTASTXSTYTAXTAY"
19044 TA$(2) = TA$(2)+"TSXTXATXSTYA???"
19046 RETURN
```

Queste tabelle apparentemente scoraggianti sono in realtà assai semplici una volta che si sia ben compresa la spiegazione generale data sopra del lavoro di un disassembler.

Le tre sezioni della tabella definite fra le linee 19011 e 19029 vengono usate per creare, attraverso chiamate al modulo precedente, una riga della matrice TA$, contenente caratteri il cui codice cada nell'intervallo 0-56. Questi valori puntano ad una tabella seguente contenente i nomi, in linguaggio assemblatore, dei 56 tipi di codice operativo disponibili quando si esprima un'istruzione macchina in linguaggio assemblatore, più un codice usato per segnalare che è stato rilevato un codice operativo non valido. Nel codice macchina del 6502/6510 esistono più di 150 codici operativi possibili, perché allora solo 56 rappresentazioni mnemoniche in linguaggio assemblatore? La risposta a questa domanda è che i codici operativi del linguaggio macchina possono essere raccolti in gruppi, ad esempio quelli che caricano un valore nell'accumulatore, che abbiano una rappresentazione mnemonica comune. All'interno di ciascun gruppo sussistono però grosse differenze fra gli operandi, cioè nel modo di reperire il valore da elaborare. Quindi ogni codice operativo sarà associato ad un unico tipo di operando, mentre un codice mnemonico potrebbe venir associato a parecchi tipi diversi di operando al momento di tradurre il programma macchina in linguaggio assemblatore.

Così un codice operativo di valore 127 avrebbe un punto di ingresso nella posizione 127 di TA$ (0). Il codice ASCII del carattere contenuto in quella posizione verrà usato per attribuire un valore fra 0 e 56. Questo valore sarà poi utilizzato per puntare a tre caratteri della sezione di tabella definita nelle linee 19940-19944. Queste cinque linee di testo, una volta suddivise in gruppi di tre caratteri, rappresentano tutte le possibili rappresentazioni mnemoniche del linguaggio assemblatore 6502/6510 per i vari codici operativi.

Le rimanenti sezioni delle tabelle, definite dalle linee fra 19030 e 19037 danno il tipo di operando associato al particolare codice operativo. I tipi di operando verranno spiegati in maggior dettaglio in seguito.

SEZIONE 2: operandi e loro tipi

Come detto nella prefazione, questo libro non intende fornire un'introduzione al linguaggio macchina del 6502/6510. Si presuppone che chi voglia usare questo libro sia già in parte a conoscenza di un certo numero di concetti che stanno alla base della programmazione in linguaggio macchina o in linguaggio assemblatore oppure che questo libro venga usato insieme ad un testo introduttivo generale sul linguaggio assemblatore del 6502/6510 che spieghi le varie funzioni disponibili sul chip in questione. È comunque necessario a questo punto per la comprensione del programma fornire qualche breve spiegazione sul modo in cui il chip 6502/6510 interpreta gli operandi, cioè i valori delle locazioni di memoria su cui è in grado di compiere le sue 56 operazioni.

Il chip 6502/6510 è in grado di riconoscere 11 differenti metodi, noti come modi di indirizzamento, per ottenere il valore su cui operare in un programma in codice macchina. Ogni singolo codice operativo richiede l'uso di uno di questi 11 metodi diversi. Il programma disassembler dev'essere in grado di riconoscere il codice operativo e quindi di estrarre da esso il tipo di indirizzamento da utilizzare.

Le due forme più semplici di indirizzamento sono l'indirizzamento in accumulatore e l'indirizzamento implicito:

1) Indirizzamento in accumulatore: alcuni codici operativi specificano, senza esplicitare ulteriormente un valore o un indirizzo di memoria, che l'operazione da compiere deve essere eseguita sul contenuto del registro accumulatore della CPU. Un esempio di, questo tipo di indirizzamento può essere lo 'shift left accumulator' (SLA in linguaggio assemblatore), che fa scorrere i bits 0-6 dell'accumulatore verso sinistra, moltiplicando così per due il valore rappresentato da quei bit. Quando si specifica un'operazione di questo tipo non c'é bisogno di ulteriori precisazioni ed è necessario un solo bit di memoria per rappresentare un'istruzione di questo tipo in un programma di codice macchina.

2) Indirizzamento implicito: l'indirizzamento in accumulatore è un caso particolare di questo modo di indirizzamento. Ci sono altri codici operativi che specificano implicitamente il luogo in cui trovare il valore da elaborare. Un esempio di questi può essere 'trasferisci l'accumulatore al registro Y' (TAY). L'effetto di questa operazione è esattamente quello enunciato e non c'è altra necessità di dire come ottenere il

valore da trasferire o dove dovrà essere posto. Anche questa è un'istruzione macchina da un byte.

3) Indirizzamento immediato: i codici operativi che impiegano questo tipo di indirizzamento richiedono che il valore su cui operare venga specificato insieme al codice operativo stesso, per mezzo di valori compresi fra 0 e 255, o del possibile contenuto di un singolo byte di memoria. Un esempio di codice operativo di questo tipo è 'load accumulator immediate' (LDA). Un'istruzione contenente questo codice operativo potrebbe essere LDA 127. L'effetto di questa istruzione sarebbe quello di caricare nell'accumulatore il valore 127. Tradotto in codice macchina questo tipo di istruzioni richiede un byte di memoria per specificare il codice operativo ed un altro byte per specificare il valore su cui operare.

4) Indirizzamento relativo: viene impiegato quando in un programma è necessario utilizzare dei salti ed è richiesto un valore per specificare la locazione di memoria cui il programma dovrà saltare durante l'esecuzione. Come per il tipo precedente di indirizzamento, il valore dovrà essere nell'intervallo 0-255 ma questo intervallo è suddiviso in una metà positiva ed una negativa, con i valori 0-127 che provocano un salto positivo ed i valori 128-255 uno negativo (viene sottratto 127 dal valore in oggetto). Il salto è misurato relativamente all'indirizzo del byte seguente l'istruzione di salto. Un esempio di questo genere di codice operativo può essere 'branch nonzero' (BNE). Un'istruzione contenente questo codice operativo potrebbe prendere la forma BNE 127, l'effetto dell'istruzione sarebbe quello di saltare in avanti di 127 byte prima di riprendere l'esecuzione del programma nel caso che l'operazione precedente del programma avesse dato un risultato diverso da zero. L'indirizzamento relativo, come i tipi precedenti, impiega un byte per il codice operativo ed un byte per l'operando.

Prima di discutere i restanti modi di indirizzamento è necessario comprenderne due tipi che non sono implementati singolarmente ma formano uno la base dell'altro:
a) Indirizzamento indicizzato: questo metodo impiega uno dei due registri del chip 6502/6510 noti come 'registri indice'.

Questo tipo di codice operativo usa un operando che specifica un indirizzo di memoria ma, prima che questo indirizzo venga utilizzato, esso o il suo contenuto vengono modificati aggiungendovi il contenuto presente di uno dei registri indice. Allora un'istruzione che usa l'indirizzamento indicizzato richiede che:
I) ci sia un valore nel registro indice
II) l'operando specifichi un indirizzo di memoria.
b) Indirizzamento in pagina zero: si riferisce al fatto che sebbene il chip 6502/6510 abbia solo cinque registri (locazioni nel chip in cui si possano facilmente porre ed e-

laborare valori) accessibili al programmatore in linguaggio macchina, questa limitazione al confronto di altri popolari chip di CPU è superata dalla possibilità di considerare l'intera area di memoria dall'indirizzo 0 all'indirizzo 255 come una serie di 128 registri a due byte che possono venir utilizzati dalla CPU per operazioni varie. L'indirizzamento in pagina zero è il modo di indirizzamento che rende accessibile quest'area di memoria.

Tornando ora ai principali modi di indirizzamento forniti dal chip 6502/6510 troviamo:

5) Indirizzamento indicizzato in pagina zero: in questa forma di indirizzamento sono combinati i due metodi presentati sopra. In un'istruzione di questo tipo il valore contenuto nel registro indice potrebbe essere sette, nel qual caso l'operando a due byte si riferirebbe ad un indirizzo in pagina zero della memoria (0-255), a cui andrebbe sommato il contenuto dello specificato indice.

6) Indirizzamento indiretto: qui l'operazione specificata dal codice operativo verrà eseguita su di un indirizzo non direttamente espresso nell'istruzione assembler, ma contenuto nei due bytes al cui indirizzo punta l'operando a due byte. Un esempio di istruzione di questo genere può essere 'jump' (JMP). Questo codice operativo sarebbe seguito da un operando a due byte. L'operando non è l'indirizzo di memoria cui il programma in esecuzione dovrebbe saltare, piuttosto i due byte che iniziano all'indirizzo specificato contengono un indirizzo. È questo secondo indirizzo quello cui il programma dovrà saltare. Quindi JMP ($AAAA) non indica un salto all'indirizzo AAAA, ma all'indirizzo rappresentato dal valore memorizzato nei due byte di memoria $AAAA ed $AAAB.

Altre due forme di indirizzamento indiretto disponibili sul chip 6502/6510 usano il concetto di indirizzamento indicizzato descritto sopra:

7) Indirizzamento pre-indicizzato: come per un normale indirizzamento indiretto, gli operandi di questo tipo contengono indirizzi ai quali si troveranno i valori da elaborare. Però prima di ottenere l'indirizzo iniziale vengono aggiunti operandi pre-indicizzati al contenuto del registro X della CPU. Così se il registro X contiene $100 e l'operando è $100, l'indirizzo in cui si cercherà il valore desiderato è $200.

8) Indirizzamento post-indicizzato: qui l'operando specifica una locazione di memoria dalla quale viene prelevato il contenuto che si somma a quello del registro Y della CPU. Il risultato è un indirizzo sul cui contenuto si eseguirà l'elaborazione.

9) Indirizzamento assoluto: in questo caso l'operando a due byte specifica un indirizzo di memoria dove si troverà il valore su cui operare. Ad esempio l'istruzione

'load accumulator' usata con questo tipo di indirizzamento potrà avere la forma LDA $AAAA, che avrebbe l'effetto di caricare nell'accumulatore il valore contenuto nel byte $AAAA della memoria.

10 e 11) Indirizzamento assoluto X ed assoluto Y: in questi due tipi l'indirizzo specificato nell'operando a due byte viene sommato al contenuto del registro X o Y per ottenere l'indirizzo finale del valore da elaborare. Se ad esempio il contenuto del registro X è $5 e l'operando è $AAAA, l'indirizzo del valore da elaborare per esempio con un'operazione del tipo LDA $AAAA,X sarebbe contenuto nel byte $AAAF della memoria da cui verrebbe passato nell'accumulatore.

Date queste brevi spiegazioni sui diversi tipi di operandi che il chip 6502/6510 è in grado di riconoscere, dovreste trovare abbastanza facili da seguire senza troppi commenti le sezioni del programma che si occupano di creare le istruzioni in linguaggio assemblatore a partire dalle equivalenti in linguaggio macchina. Nei moduli seguenti, quando un codice operativo è prelevato dalla memoria, il programma ricaverà il corretto modo di indirizzamento per quel codice operativo accedendo alle tabelle memorizzate nella sezione precedente, ottenendo un valore che registrerà nella variabile OP (operando). Il valore di OP tradotto in un metodo di indirizzamento è riportato nella tavola qui sotto, cui troverete utile far ricorso seguendo i moduli del programma disassembler.

VALORE DI 'OP'	**MODI DI INDIRIZZAMENTO**
0	Accumulatore
1	Implicito
2	Immediato
3	Relativo
4	Indicizzato pag. 0, X
5	Indicizzato pag. 0, Y
6	Pag. zero
7	Pre-indicizzato indiretto (X)
8	Post-indicizzato indiretto (Y)
9	Assoluto indiretto
10	Assoluto indicizzato, X
11	Assoluto indicizzato, Y
12	Assoluto

TAVOLA DI CONTROLLO

```
19000 123      19001 139      19002 123
19005 116      19007 132      19010 228
19011 136      19012 89       19013 78
19014 90       19015 91       19016 116
19017 93       19018 241      19019 164
19020 130      19021 154      19022 226
19023 193      19024 61       19025 87
19026 213      19027 189      19028 58
19029 213      19030 162      19031 141
19032 118      19033 108      19034 111
19035 147      19036 125      19037 11
19038 75       19040 84       19041 175
19042 10       19043 82       19044 238
19046 142
```

MODULO 2.3

```
15450 REM********************************
15451 REM ACCUMULATOR (OP=0)
15452 REM********************************
15460 O1$ = O1$+"A"
15500 REM IMPLIED (OP=1)
15510 RETURN
```

Questo breve modulo si occupa dei due modi di indirizzamento più semplici:
a) indirizzamento in accumulatore: tutto ciò che si richiede per disassemblare questo tipo è l'aggiunta di 'A' al codice operativo standard.
b) indirizzamento implicito: qui il codice operativo stesso sottintende il suo operando, quindo non c'è bisogno di alcuna azione.

TAVOLA DI CONTROLLO

```
15450 123      15451 108      15452 123
15460 105      15500 49       15510 142
```

MODULO 2.4

```
15550 REM********************************
15551 REM IMMEDIATE (OP=2)
15552 REM********************************
15560 GOSUB 11100
15570 O1$ = O1$+"#$"+H$
15580 RETURN
```

Questo modulo si occupa dell'indirizzamento immediato. Il byte seguente il codice operativo viene interpretato come un operando nell'intervallo 0-255.

TAVOLA DI CONTROLLO

```
15550 123      15551 189      15552 123
15560 160      15570 133      15580 142
```

MODULO 2.5

```
15600 REM********************************
15601 REM RELATIVE (OP=3)
15602 REM********************************
15610 GOSUB 11100
15620 IF H>127 THEN H = H-256
15630 H = H+AD
15640 GOSUB 11000
15650 O1$ = O1$+"$"+H$
15660 RETURN
```

Questo modulo si occupa dell'indirizzamento relativo e trasforma il byte che segue il codice operativo in un numero nell'intervallo $-128/+127$.

TAVOLA DI CONTROLLO

```
15600 123      15601 139      15602 123
15610 160      15620 239      15630 177
15640 159      15650 98       15660 142
```

```
15300 REM********************************
15301 REM ADD OPERAND IN OP TO O1$
15302 REM********************************
15310 ON OP+1 GOTO 15450,15500,15550,156
00
15330 IF OP>6 AND OP<10 THEN O1$ = O1$+"
("
15340 GOSUB 11100
15350 O1$ = O1$+"$" : T$ = H$
15360 IF OP<9 THEN 15390
15370 GOSUB 11100
15380 O1$ = O1$+H$
15390 O1$ = O1$+T$
15400 IF OP=9 OR OP=8 THEN O1$ = O1$+")"
15410 IF OP-INT(OP/3)*3=1 THEN O1$ = O1$
+",X"
15420 IF OP-INT(OP/3)*3=2 THEN O1$ = O1$
+",Y"
15430 IF OP=7 THEN O1$ = O1$+")"
15440 RETURN
```

Questa semplice sezione di frasi IF regola il formato delle istruzioni in linguaggio assembler a seconda dei diversi modi di indirizzamento. Il modo migliore di capire questa sezione è quello di confrontare ciò che essa fa all'operando sulla base dei valori di OP presentati nella tabella più sopra.

TAVOLA DI CONTROLLO

```
15300 123     15301 158     15302 123
15310 110     15330 10      15340 160
15350 156     15360 31      15370 160
15380 80      15390 92      15400 230
15410 207     15420 209     15430 107
15440 142
```

SEZIONE 3: disassemblaggio della memoria

Abbiamo introdotto le sezioni del programma che permettono di tradurre da linguaggio macchina in linguaggio assemblatore. Resta ora da aggiungere l'insieme

dei moduli che permettono al programma di prelevare il contenuto di una specifica area di memoria del 64 in modo che le istruzioni in esse contenute in linguaggio macchina possano essere disassemblate e visualizzate sullo schermo.

MODULO 2.7

```
15700 REM********************************
15701 REM DISASSEMBLE INSTRUCTION
15702 REM********************************
15710 O2$ = ""
15715 GOSUB 11100 : H = H+1
15720 IF H>255 THEN H = 3
15730 T = ASC (MID$ (TA$(0),H,1))
15750 O1$ = MID$ (TA$(2),T*3+1,3)+" "
15760 OP = ASC (MID$ (TA$(1),INT((H+1)/2
),1))
15770 IF (H AND 1) =1 THEN OP = OP/16
15780 OP = OP AND 15
15790 RETURN
```

TAVOLA DI CONTROLLO

```
15700 123     15701 93      15702 123
15710 219     15715 119     15720 148
15730 131     15750 148     15760 224
15770 146     15780 133     15790 142
```

Questo modulo costruisce l'istruzione in linguaggio assemblatore sulla base delle informazioni ricavate dalla memoria.

Commento

15715-15720: ottenuto il byte del codice operativo, il suo valore viene posto nella variabile 'H'.
15730: il codice operativo viene usato per ottenere un puntatore da TA$(0) che indicherà la posizione in TA$(2) del formato a tre lettere in linguaggio assemblatore di quel codice.

15750: viene aggiunto uno spazio dopo il codice operativo per uniformarlo al formato standard del linguaggio assembler.
15760-15780: viene ricavato dalla tabella TA$(I) il modo di indirizzamento associato al codice operativo.

MODULO 2.8

```
15800 REM********************************
15801 REM DISASSEMBLE MEMORY AREA
15802 REM********************************
15810 GOSUB 12050
15820 PRINT "[CLH]" : FOR I = 1 TO 20
15825 H = AD : GOSUB 11000 : PRINT H$ TA
B(6) ;
15830 GOSUB 15700 : GOSUB 15300
15850 PRINT O2$ TAB(14) O1$
15860 NEXT I
15865 PRINT
15870 GOSUB 11850
15880 IF CO THEN 15820
15890 RETURN
```

TAVOLA DI CONTROLLO

```
15800 123     15801 13      15802 123
15810 165     15820 93      15825 244
15830 202     15850 115     15860 235
15865 153     15870 172     15880 36
15890 142
```

Questo è il modulo che controlla il formato delle istruzioni in linguaggio assemblatore ottenute dai moduli precedenti. Per una spiegazione delle varie chiamate delle subroutine, si veda in Appendice la Tabella delle funzioni delle subroutine.

Sommario

Anche se state lavorando su questo libro con un buon testo base sul 6502/6510 vale la pena, a questo punto, di passare un po' di tempo giocando col vostro assembler e monitor. Provate a disassemblare alcune delle routine della ROM del 64 e cercate di capire come funzionano. Alcuni indirizzi interessanti per iniziare a disassemblare sono elencati sotto, con le funzioni delle routine che vi si trovano.

Fate attenzione comunque che un disassembler è utile solo se viene fornita una posizione di partenza corretta in memoria. Se iniziate a disassemblare la memoria in un punto a metà di un'istruzione macchina, i primi byte almeno del listing disassemblato saranno privi di senso, dal momento che parti di operandi saranno state tradotte come codici operativi. Alla fine, dopo aver saltato un numero di istruzioni apparentemente errate o addirittura insensate, il disassembler incontrerà istruzioni corrette. Da qui in avanti verrà disturbato solo dalla presenza in memoria di tabelle, che tenterà nuovamente di tradurre come se si trattasse di istruzioni in linguaggio macchina. Quando vi imbattete in tali problemi l'unica soluzione è quella di spostare l'indirizzo iniziale finchè non sia terminata la tabella e vengano trovate istruzioni sensate all'inizio del listing disassemblato.

Qui sotto c'è un esempio di disassemblaggio di un'area dell'interprete del 64 iniziante all'indirizzo di una routine la cui funzione è quella di accettare in ingresso una nuova linea BASIC usando varie subroutine del monitor e del 'kernal' del 64.

```
A480      6C0203    JMP ($0302)
A483      2060A5    JSR $A560
A486      867A      STX $7A
A488      847B      STY $7B
A48A      207300    JSR $0073
A48D      AA        TAX
A48E      F0F0      BEQ $A480
A490      A2FF      LDX #$FF
A492      863A      STX $3A
A494      9006      BCC $A49C
A496      2079A5    JSR $A579
A499      4CE1A7    JMP $A7E1
A49C      206BA9    JSR $A96B
A49F      2079A5    JSR $A579
A4A2      840B      STY $0B
A4A4      2013A6    JSR $A613
A4A7      9044      BCC $A4ED
A4A9      A001      LDY #$01
A4AB      B15F      LDA ($5F),Y
A4AD      8523      STA $23

CONTINUE ( Y/N ) :
```

CAPITOLO 3
FILE EDITOR MASTERCODE

Prima di procedere con la parte principale del programma assemblatore esamineremo il file editor, che permette di immettere in forma opportuna i programmi in linguaggio macchina e di editarli convenientemente.

Discutendo del disassembler abbiamo già notato il formato di alcune singole istruzioni che verranno usate nei programmi in linguaggio assemblatore. Se avete usato il disassembler per tradurre parte della ROM del 64, avrete anche visto in quale forma i programmi in linguaggio assemblatore vengono usualmente presentati, che consiste di tre tipi d'informazione per ciascuna istruzione:

1) l'indirizzo di memoria dove l'istruzione si trova

2) il contenuto, in esadecimale, dei byte che la compongono

3) la forma in linguaggio assemblatore dell'istruzione

Quando si introduce un programma in linguaggio assemblatore per mezzo di un assembler, sono necessarie le sole istruzioni in linguaggio assemblatore. Ci sono però alcuni problemi connessi alla semplice introduzione di una lunga lista di istruzioni in linguaggio assemblatore. Ad esempio cosa succede se dopo aver immesso un lungo programma in linguaggio assemblatore ci accorgiamo che dobbiamo inserire alcune istruzioni nel mezzo di esso oppure che dobbiamo cancellarne alcune? Dobbiamo ripetere tutto il programma correttamente? Ovviamente l'ideale sarebbe qualcosa del tipo offerto dall'interprete BASIC del 64 - linee numerate che vengono automaticamente cancellate o inserite al posto giusto, con la possibilità di modificare linee dovunque nel programma. È compito del file editor offrire queste possibilità, benché la sezione di programma presentata qui vada oltre, permettendo anche di renumerare il programma e di salvare (o caricare) il file in linguaggio assemblatore prima che l'assemblatore proceda effettivamente ad elaborarlo e tradurlo in codice macchina.

Va sottolineato il fatto che il file editor non è una parte dell'assemblatore propriamente detto poiché non si occupa del tipo di materiale che viene immesso, ma permette semplicemente di inserire in un file delle linee numerate. Nulla viene controllato o elaborato finché l'assemblatore vero e proprio non viene attivato.

SEZIONE 1: inizializzazioni

MODULO 3.1

```
24800 REM********************************
24801 REM FILE EDITOR MENU
24802 REM********************************
24820 PRINT "[CLH][GRN] ------------ FIL
E EDITOR ------------[BLU][CDW]"
24835 PRINT "       0) EXIT FROM FILE EDI
TOR"
24840 PRINT "       1) INPUT LINE(S)"
24850 PRINT "       2) LIST LINE(S)"
24860 PRINT "       3) DELETE LINE(S)"
24870 PRINT "       4) RENUMBER FILE"
24880 PRINT "       5) INITIALISE FILE"
24890 PRINT "       6) LOAD FILE"
24900 PRINT "       7) SAVE FILE"
24910 PRINT "       8) ADD MACHINE CODE T
O FILE"
24915 PRINT "       9) CHANGE DEVICE NUMB
ER[CDW][CDW][CDW][CDW][CDW]"
24920 INPUT " COMMAND ( 0-9 ) : "; CO
24940 IF CO=0 THEN RETURN
24950 IF CO>0 THEN ON CO GOSUB 24600,244
00,24500,24700,24300,23600,23700,25000
24960 IF CO>8 THEN ON CO-8 GOSUB 25500
24970 GOTO 24800
```

Un modulo completamente dedicato al menu.

TAVOLA DI CONTROLLO

```
24800 123     24801 11      24802 123
24820 125     24835 235     24840 179
24850 96      24860 216     24870 218
24880 102     24890 156     24900 172
24910 90      24915 243     24920 182
24940 148     24950 30      24960 252
24970 167
```

MODULO 3.2

```
24300 REM********************************
24301 REM INITIALISE FILE
24302 REM********************************
24310 PTR$ = "" : E$ = "" : FOR X = 0 TO
 254 : E$ = E$+CHR$ (X) : NEXT : RETURN
```

Questo modulo predispone le variabili necessarie per trattare un nuovo file — richiamando questa opzione con un file già in memoria si provocherà la perdita di questo file. Le due principali variabili sono **PTR$**, che indicherà la posizione corretta degli elementi del file, ed **E$**, che terrà conto della posizione degli spazi per nuovi inserimenti. L'uso di **PTR$** verrà descritto nel modulo 6.

TAVOLA DI CONTROLLO

```
24300 123     24301 218     24302 123
24310 217
```

MODULO 3.2A

```
19980 DIM FI$(254) : GOSUB 24300
```

Questo modulo fa parte in effetti della routine di inizializzazione principale per le tabelle del disassembler. La sua funzione è quella di predisporre la matrice del file

principale (FI$) quando il programma viene eseguito per la prima volta. Una volta che il programma è partito, la matrice viene reinizializzata richiamando il modulo precedente.

TAVOLA DI CONTROLLO

```
19980 101
```

SEZIONE 2: Immissione di linee

MODULO 3.3

```
24600 REM********************************
24601 REM INPUT LINE(S)
24602 REM********************************
24610 PRINT "[CLH]"
24620 IN$ = "" : INPUT IN$ : GOSUB 24000
 : IF LN=-65536 THEN 24665
24650 GOSUB 23900 : IF LEN(IN$)=0 THEN 2
4680
24660 GOSUB 23100 : IF NOT ERR THEN 2462
0
24665 RETURN
24680 GOSUB 23020 : IF NOT ERR THEN GOSU
B 23300
24690 GOTO 24620
```

Questo è il modulo che, all'immissione di una linea, attribuisce i compiti necessari alle varie routine del file editor. Oltre a distribuire i compiti ad altri moduli, la sua unica funzione è quella di permettere l'immissione della linea nella forma di IN$ e di determinare se viene inserito un numero di linea senza alcuna linea, cioè se si vuole eseguire una cancellazione.

TAVOLA DI CONTROLLO

```
24600 123     24601 43      24602 123
24610 144     24620 251     24650 108
24660 94      24665 142     24680 6
24690 167
```

MODULO 3.4

```
24000 REM********************************
24001 REM GET LINE NUMBER
24002 REM********************************
24010 LN = -65536
24020 IF LEN(IN$)=0 OR IN$<"0" OR LEFT$
(IN$,1)>"9" THEN 24090
24030 FOR T = 1 TO LEN(IN$)
24040 IF MID$ (IN$,T,1)<="9" AND MID$ (I
N$,T,1)>="0" THEN NEXT T
24080 LN = VAL(LEFT$ (IN$,T-1)) : IN$ =
MID$ (IN$,T)
24090 RETURN
```

Ottenuto un input nella forma **IN$**, si ricava un numero di linea dall'inizio della stringa. La stringa viene esaminata carattere per carattere per trovare il primo di essi al di fuori dell'intervallo 0-9, quindi si calcola il VAL della stringa fino a quel punto. Le stringhe che non iniziano con un numero di linea provocano un'attribuzione del valore –65536 al numero stesso (LN), segnalando così l'errore, in caso contrario il numero di linea viene memorizzato in LN ed i caratteri che lo rappresentano vengono sottratti alla stringa originaria.

TAVOLA DI CONTROLLO

```
24000 123     24001 192     24002 123
24010 64      24020 99      24030 203
24040 150     24080 79      24090 142
```

MODULO 3.5

```
23900 REM********************************
23901 REM REMOVE LEADING SPACES
23902 REM********************************
23910 FOR T = 1 TO LEN(IN$)
23920 IF MID$ (IN$,T,1)=" " THEN NEXT T
23950 IN$ = MID$ (IN$,T) : RETURN
```

La stringa **IN$**, privata del suo numero di linea, può iniziare con uno o più spazi — questo modulo li rimuove.

TAVOLA DI CONTROLLO

```
23900 123    23901 112    23902 123
23910 203    23920 81     23950 11
```

MODULO 3.6

```
23000 REM*******************************
23001 REM FILE EDITOR
23002 REM*******************************
23010 REM FILE EDITOR
23020 REM FIND LINE NUMBER IN 'LN' IN FI
LE
23030 T = LEN(PTR$)+1 : T2 = -1
23040 T = T-1 : IF T<=0 THEN GOTO 23080
23050 T1 = ASC (MID$ (PTR$,T,1))
23060 T2 = ASC (MID$ (FI$(T1),1,1))+256*
ASC (MID$ (FI$(T1),2,1))
23070 IF T2>LN THEN 23040
23080 ERR = NOT(T2=LN) : IF ERR THEN T =
 T+1
23090 RETURN
```

Prima di procedere col modulo che inserisce effettivamente una linea nel file, dobbiamo occuparci di questo, la cui funzione è determinare la posizione corretta per la nuova linea (ammesso che essa abbia un numero di linea valido). Esaminando il modulo comprenderemo meglio l'uso di **PTR$**.

Commento

23030: nella ricerca della posizione corretta, in cui inserire una linea faremo uso della stringa che abbiamo chiamato **PTR$** abbreviazione di 'pointer string' (stringa puntatore). Una stringa puntatore è un metodo standard di superare i problemi di inserimento di nuove linee in matrici a più righe. Non che la cosa sia difficile, ma semplicemente l'inserimento di una nuova linea all'inizio di quella che potenzialmente potrebbe essere una matrice anche di 250 righe provoca lo scorrimento di tutte le righe presenti, un compito che può consumare tempo e creare anche problemi di riorganizzazione rallentando ulteriormente le cose. Usando una stringa pun-

tatore si può superare tutto questo, dal momento che il contenuto della matrice non viene mai fatto scorrere. Tutte le manipolazioni vengono eseguite su un'unica stringa. Invece di cercare la posizione corretta nella matrice e poi di far scorrere tutto per fare spazio alla nuova linea, ciò che faremo è trovare quale dovrebbe essere la posizione corretta (in base al numero di linea), piazzare la linea da inserire nel primo spazio libero che si trovi e quindi indicarne la posizione effettiva al punto giusto della stringa puntatore.

In questo modo la stringa puntatore potrebbe contenere una serie di byte con i valori 34,76,233,176... Ciò che questo significherebbe è che la vera prima linea della lista si trova nella posizione 34, la seconda nella posizione 76, la terza nella posizione 233 e così via. Per accedere in ordine alla matrice di linee dobbiamo prima esaminare PTR$, ricavarne la posizione della prima linea, quindi esaminare il secondo carattere di PTR$ per trovare la posizione della seconda. Dal momento che abbiamo accettato un limite arbitrario di 255 linee per ogni file, tutti i puntatori possono venir rappresentati sotto forma di un singolo carattere in PTR$ – i caratteri hanno un valore ASCII compreso fra 0 e 255. Per eseguire un inserimento, sarà necessario soltanto suddividere PTR$ in due parti e piazzarvi nel mezzo un nuovo indicatore con un risparmio di tempo considerevole. In questa linea in particolare la variabile principale di ricerca (T) è inizializzata al valore LEN(PTR$) +1, in modo che la ricerca inizi al termine di PTR$.

23050: il valore del carattere T in PTR$ è la posizione di quella che dovrebbe essere la linea T nella matrice (non la linea col numero di linea T, ma quella nella posizione T se contassimo dall'inizio del file).

23060: ottiene il numero di linea della linea memorizzata in FI$ alla posizione TI.

23070: continua la ricerca finché viene trovato un numero di linea maggiore di quello della linea da inserire (LN).

23080: viene posta uguale a –1 la variabile ERR se il numero di linea da inserire non è uguale a quello di una linea già presente nel file. Questo permette al modulo successivo di sapere se una linea viene inserita o riscritta.

TAVOLA DI CONTROLLO

```
23010 182      23020 183      23030 29
23040 17       23050 160      23060 86
23070 92       23080 16       23090 142
```

```
23100 REM*********************************
23101 REM ADD LINE TO FILE
23102 REM*********************************
23105 IF LN<0 OR LN>65535 THEN 23215
23110 GOSUB 23020
23120 IF NOT ERR THEN T1 = ASC (MID$ (PT
R$,T,1)) : GOTO 23150
23130 IF E$="" THEN ERR = TRUE : GOTO 23
220
23140 T1 = ASC (E$)  : E$ = MID$ (E$,2)
23150 T2 = INT(LN/256)
23160 FI$(T1) = CHR$ (LN-T2*256)+CHR$ (T
2)+IN$
23170 IF NOT ERR THEN 23220
23180 T$ = "" : T1$ = ""
23190 IF T>1 THEN T$ = LEFT$ (PTR$,T-1)
23200 IF T<=LEN(PTR$) THEN T1$ = MID$ (P
TR$,T)
23210 PTR$ = T$+CHR$ (T1)+T1$
23215 ERR = FALSE
23220 RETURN
```

Questo è il modulo che esegue l'effettivo inserimento della linea nel file.

Commento

23105: usando due byte, 0-65535 è il massimo intervallo possibile per numeri di linea.

23120: se il modulo precedente restituisce un errore, significa che si sta immettendo una linea con un nuovo numero. Se non c'é errore la linea da immettere è semplicemente una riscrittura di una linea esistente e **PTR$** non ha bisogno di essere modificato.

23130: per accelerare ancora il processo di inserimento, viene utilizzata una seconda stringa (E$) per registrare tutti gli spazi esistenti nel file. Invece di ricercare il primo spazio libero, la linea verrà inserita nella posizione indicata dal primo carattere in E$-carattere che verrà eliminato finchè non tornerà libera la posizione corrispondente.

23150-23160: vengono creati i byte dei numeri di linea a partire da LN e la nuova linea viene inserita. Notate che avendo messo il byte 'alto' nella variabile T2 (=LN/256), non c'è bisogno di porre un'altra variabile uguale a LN-256*INT(LN/256). Mettendo semplicemente LN in forma ASCII si perderà tutto ciò che è superiore a 255, come se si fosse eseguito un AND fra LN e 255.

23170-23210: se stiamo trattando un nuovo numero di linea, allora si deve aggiungere un carattere a PTR$. La posizione del carattere è indicata da T ed è sufficiente prendere LEFT$ (PTR$, T−1) e MID$(PTR$,T) per inserire fra queste il carattere opportuno.

TAVOLA DI CONTROLLO

```
23100 123      23101 195      23102 123
23105 79       23110 164      23120 1
23130 40       23140 206      23150 98
23160 84       23170 60       23180 7
23190 193      23200 201      23210 30
23215 70       23220 142
```

MODULO 3.8

```
23300 REM********************************
23301 REM DELETE LINE POINTED AT BY T
23302 REM********************************
23310 T$ = "" : T1$ = ""
23320 IF T>1 THEN T$ = LEFT$ (PTR$,T-1)
23330 IF T<LEN(PTR$) THEN T1$ = MID$ (PT
R$,T+1)
23340 E$ = E$+MID$ (PTR$,T,1)
23350 PTR$ = T$+T1$
23360 RETURN
```

Potrebbe sembrare strano descrivere questo modulo nel paragrafo dedicato all'inserimento delle linee, dal momento che il suo scopo è quello di eliminarle. La ragione per cui ne parliamo qui è che, inserendo nuove linee, ne viene immessa una composta dal solo numero di linea, la riga corrispondente viene cancellata, analogamente a quanto succede in BASIC. Il modulo è perciò chiamato dal modulo di

controllo principale per l'inserimento. La procedura seguita è l'immagine speculare di quella usata nell'inserimento, con la rimozione di un carattere puntatore da PTR$ e la registrazione della posizione della linea come spazio libero in E$. Notate che non c'è bisogno di eliminare fisicamente il contenuto della linea che permane, ma esso non verrà più riconosciuto dal file editor e verrà riscritto non appena si inserirà una nuova linea.

TAVOLA DI CONTROLLO

```
23300 123      23301 193      23302 123
23310 7        23320 193      23330 242
23340 128      23350 215      23360 142
```

SEZIONE 3: listing e cancellazione

MODULO 3.9

```
24200 REM********************************
24201 REM  FIRST AND LAST LINES
24202 REM********************************
24205 IN$ = "" : INPUT "FIRST - LAST LIN
ES : "; IN$
24210 SL = 0 : FL = 65535 : T3 = 0 : ERR
 = FALSE
24220 IF LEN(IN$)=0 THEN 24295
24230 GOSUB 24000
24240 IF LN>=0 THEN SL = LN : GOTO 24260
24250 IF LN>-65536 THEN FL = -LN : GOTO
24295
24260 GOSUB 23900 : IF LEN(IN$)=0 THEN F
L = SL : GOTO 24295
24270 IN$ = MID$ (IN$,2) : GOSUB 23900
24290 IF LEN(IN$)>0 THEN GOSUB 24000 : F
L = LN
24295 ERR = SL<0 OR SL>65535 OR FL<0 OR
FL>65535 OR ERR : RETURN
```

Questo modulo è usato nel listing e nella cancellazione di blocchi per ottenere in ingresso coppie di numeri di linea nella forma '100-300'.

24210-24220: linea iniziale (SL) e linea finale (FL) vengono inizializzate con i valori estremi permessi. Se l'utente preme semplicemente RETURN all'apparire del 'prompt', l'intero file verrà listato dall'inizio alla fine.

24230-24250: il primo numero di linea viene ricavato dalla subroutine in linea 24000. Se è maggiore di zero SL viene posto uguale ad esso. Se l'ingresso fosse del tipo '–300' questo verrebbe restituito come meno 300. In questo caso SL resta a zero, ma FL verrà posto uguale a 300 ed il file verrà listato fino alla linea 300.

24260: tutti gli spazi iniziali vengono eliminati da ciò che resta in IN$ dopo la rimozione del primo numero. Se non è rimasto nulla allora FL è posto uguale ad SL e viene listata una sola linea.

24270-24290: IN$ viene privata del '–' precedente il secondo numero, tutti gli spazi precedenti vengono rimossi ed il secondo valore viene ottenuto.

TAVOLA DI CONTROLLO

```
24200 123     24201 57      24202 123
24205 168     24210 170     24220 73
24230 163     24240 11      24250 131
24260 180     24270 6       24290 125
24295 38
```

MODULO 3.10

```
23400 REM********************************
23401 REM LIST LINES POINTED AT BY T
23402 REM********************************
23410 PRINT ASC (MID$ (FI$(T),1,1))+256*
ASC (MID$ (FI$(T),2,1)) TAB(6) ;
23420 PRINT MID$ (FI$(T),3)
23430 RETURN
```

Questo modulo stampa una linea la cui posizione è indicata dalla variabile T. Il numero di linea è ricavato dai primi due caratteri della linea stessa, quindi viene visualizzato il resto della linea.

TAVOLA DI CONTROLLO

```
23400 123      23401 157      23402 123
23410 178      23420 139      23430 142
```

MODULO 3.11

```
23500 REM********************************
23501 REM START AND FINISH POINTERS
23502 REM********************************
23510 LN = SL : GOSUB 23020
23520 SP = T
23530 LN = FL : GOSUB 23020
23540 FP = T
23545 IF ERR THEN FP = FP-1
23550 IF FP>LEN(PTR$) THEN FP = LEN(PTR$
)
23560 RETURN
```

Usando i numeri delle linee iniziali e finali questo modulo estrae da PTR$ i puntatori alla prima e all'ultima linea da listare e li pone in SP ed FP.

TAVOLA DI CONTROLLO

```
23500 123      23501 165      23502 123
23510 73       23520 233      23530 60
23540 220      23545 117      23550 189
23560 142
```

MODULO 3.12

```
24400 REM********************************
24401 REM LIST LINES
24402 REM********************************
24410 GOSUB 24200 : IF ERR THEN 24460
24420 PRINT "[CLH]" : GOSUB 23500 : IF F
P<SPOR FP=0 THEN 24460
```

```
24430 FOR T1 = SP TO FP : T = ASC (MID$
(PTR$,T1,1)) : GOSUB 23400 : NEXT : PRIN
T
24455 IF PEEK(152)=0 THEN GET T$ : IF T$
="" THEN 24455
24460 RETURN
```

Usando i puntatori iniziale e finale determinati dal modulo precedente, questo modulo richiama il modulo di visualizzazione per trasferire le linee sullo schermo. La strana linea 24445 controlla se le linee vengono al momento listate su un dispositivo come il registratore a cassette o la stampante. In caso contrario, il listing verrà visualizzato su schermo finché non verrà premuto un tasto.

TAVOLA DI CONTROLLO

```
24400 123      24401 134      24402 123
24410 154      24420 241      24430 126
24455 214      24460 142
```

MODULO 3.13

```
24500 REM********************************
24501 REM DELETE LINE(S)
24502 REM********************************
24510 GOSUB 24200 : IF ERR THEN 24460
24520 GOSUB 23500 : IF FP<SP THEN 24560
24530 T = SP : FOR T1 = SP TO FP : GOSUB
 23300 : NEXT
24560 RETURN
```

Questo è il modulo di cancellazione di un blocco. Viene incluso ora poiché la sua unica funzione è quella di richiamare i moduli precedentemente introdotti, il più voluminoso dei quali è la routine che trova la prima e l'ultima linea. Invece di listare le linee specificate il modulo di cancellazione di una linea viene richiamato a turno per

ciascuna linea. Notate che dal momento che PTR$ viene accorciato ad ogni cancellazione, il carattere cancellato ad ogni iterazione del ciclo si trova sempre nella stessa posizione.

TAVOLA DI CONTROLLO

```
24500 123     24501 78      24502 123
24510 154     24520 160     24530 179
24560 142
```

SEZIONE 4: caricamento e salvataggio

MODULO 3.14

```
23700 REM********************************
23701 REM SAVE FILE TO DEVICE
23702 REM********************************
23705 GOSUB 11250
23710 IF DEV=8 THEN IN$ = IN$+",S,W"
23715 T$ = "N" : IF DEV=8 THEN INPUT "OV
ERWRITE EXISTING FILE ( Y/N ) : "; T$
23716 IF T$="Y" THEN IN$ = "@0:"+IN$
23720 OPEN2,DEV,2,IN$ : CMD 2
23730 SL = 0 : FL = 65536
23750 GOSUB 24420 : PRINT#2 , "END"
23760 PRINT#2 : CLOSE 2
23780 RETURN
```

Questo modulo permette di salvare su disco o nastro un file che avete creato oppure di inviarlo in uscita alla stampante.

Commento

23705: routine del monitor che richiede il nome del file.
23710-23716: queste linee sono incluse a vantaggio di coloro che utilizzano l'unità a disco per memorizzare. L'effetto è di permettere all'utente di scrivere su di un file già esistente sul drive zero il file sequenziale con il contenuto di FI$. Le linee sono eseguite solo se DEV è posta uguale ad 8 (drive del disco).

23720-23760: viene aperto un file sul dispositivo specificato e l'istruzione CMD2 precisa che tutti gli output successivi saranno inviati a quel dispositivo. Non resta altro che usare le normali routine di listing per stampare tutte le linee del file, concluderle con 'END' come sigillo e infine chiudere il file.

TAVOLA DI CONTROLLO

```
23700 123    23701 177    23702 123
23705 166    23710 179    23715 32
23716 89     23720 138    23730 200
23750 116    23760 54     23780 142
```

MODULO 3.15

```
23600 REM********************************
23601 REM LOAD FILE FROM DEVICE
23602 REM********************************
23610 GOSUB 11250
23615 IF DEV=8 THEN IN$ = IN$+",S,R"
23630 OPEN2,DEV,0,IN$
23635 INPUT#2 , IN$ : IF ST THEN GOTO 23
650
23640 IF IN$<>"END" THEN GOSUB 24000 : G
OSUB 23900 : GOSUB 23100 : GOTO 23635
23650 CLOSE 2
23660 RETURN
```

Immagine speculare del modulo precedente. Notate che nel ricaricare un file da nastro o disco devono venir usate tutte le normali routine di ingresso. Ciò perché le linee sono state listate per esteso con i loro numeri di linea e non nella forma a due byte dei numeri di linea che è mantenuta correntemente in FI$. Il sistema di salvataggio/caricamento su cassetta ha difficoltà nel salvare i caratteri ASCII non stampabili e il semplice salvataggio del contenuto di FI$ provocherebbe l'alterazione di alcuni numeri di linea durante il ricaricamento.

TAVOLA DI CONTROLLO

```
23600 123    23601 51     23602 123
23610 166    23615 174    23630 31
23635 57     23640 215    23650 242
23660 142
```

MODULO 3.16

```
25500 REM********************************
25501 REM CHANGE DEVICE NUMBER
25502 REM********************************
25510 PRINT SPC(19) DEV
25520 INPUT "[CUP]NEW DEVICE NUMBER:";DE
V
25530 RETURN
```

Lo scopo di questo modulo è quello di permettere un output su cassetta, disco o stampante. Notate che un tentativo di eseguire un output o un input su di un dispositivo non presente, o di eseguire un input da un dispositivo che non è in grado di farlo, come ad esempio la stampante, può avere il risultato di bloccare il programma. I dati non andranno persi se avrete iniziato l'esecuzione del programma con un GOTO 10000 invece che con una RUN. Prima di fare ciò è utile accertarsi che il file 2 sia chiuso con l'istruzione PRINT 2: CLOSE 2 se stavate salvando al momento dell'interruzione del programma oppure con CLOSE2 se stavate caricando. Questo eviterà la possibilità di un errore del tipo "FILE ALREADY OPEN" (file già aperto).

TAVOLA DI CONTROLLO

```
25500 123     25501 14      25502 123
25510 241     25520 113     25530 142
```

SEZIONE 5: renumerazione

MODULO 3.17

```
24700 REM********************************
24701 REM RENUMBER FILE IN STEPS OF 10
24702 REM********************************
24710 LN = 10 : ERR = FALSE
24720 IF LEN(PTR$)<1 THEN 24780
24730 FOR T = 1 TO LEN(PTR$)
24735 T1 = ASC (MID$ (PTR$,T,1))
```

```
24740 FI$(T1) = CHR$ (LN-INT(LN/256)*256
)+CHR$ (LN/256)+MID$ (FI$(T1),3)
24750 LN = LN+10 : NEXT
24780 RETURN
```

Forse il modulo non merita una sezione particolare, ma esegue operazioni indipendenti da quanto visto finora. Il suo scopo è quello di renumerare il vostro file con passo 10. Ciò è realizzato privando ogni elemento introdotto, nel file dei suoi primi due caratteri e quindi ricreandoli a partire da LN, che viene incrementato di 10 ad ogni nuova linea.

TAVOLA DI CONTROLLO

```
24700 123     24701 235     24702 123
24710 173     24720 169     24730 42
24735 160     24740 94      24750 45
24780 142
```

MODULO 3.18

```
25000 REM*******************************
25001 REM ADD TO FILE FROM MEMORY
25002 REM*******************************
25010 GOSUB 12050 : GOSUB 11200 : GOSUB
24200
25050 FOR XY = AD TO EA STEP 15
25060 IN$ = " BYT " : LN = SL : SL = SL+
5
25070 FOR XZ = 0 TO 14 : 02$ = ""
25080 GOSUB 11100 : IN$ = IN$+"$"+H$
25100 IF XZ<14 AND AD<=EA THEN IN$ = IN$
+"," : NEXT XZ
25110 GOSUB 23100 : NEXT XY : RETURN
```

Dobbiamo ammettere che questo modulo è frutto di un ripensamento. La sua rilevanza non sarà del tutto chiara finché non avrete a disposizione l'assemblatore, ma ciò che fa è permettervi di specificare un'area di memoria e piazzarla in un file di programma in linguaggio assemblatore nella forma di 'direttive di byte' – il contenuto di ciascuna locazione di memoria è specificato nel file assemblatore. Non

viene eseguita alcuna modifica automatica sulle istruzioni che accedono ad indirizzi nell'area da cui il codice è stato originariamente spostato. Tali istruzioni si riferiranno ancora all'area originale di memoria.

TAVOLA DI CONTROLLO

```
25000 123     25001 200     25002 123
25010 223     25050 130     25060 78
25070 19      25080 170     25100 13
25110 120
```

Sommario

Ora che avete a disposizione il file editor è bene che ci giochiate un po' prima di procedere a immettere l'assemblatore. Questo vi aiuterà a evitare la seccatura di immettere un lungo file in linguaggio assemblatore e scoprire di averlo rovinato a causa di un uso sbagliato del file editor. Potete, se lo volete, immettere uno o due dei programmi in linguaggio assemblatore che si trovano più oltre in questo libro, salvarli su nastro o disco e poi ricaricarli per controllare se conoscete esattamente la procedura.

CAPITOLO 4
ASSEMBLER MASTERCODE

Dopo aver completato il file editor possiamo iniziare il lavoro di immissione della parte più importante e complessa del programma Mastercode, l'assembler. Il suo scopo è quello di permettersi di introdurre programmi in linguaggio assemblatore e di fornirvi strumenti per facilitare la programmazione, oltre a permettervi di tradurre automaticamente quei programmi in linguaggio macchina. Il prezzo pagato per la flessibilità e la potenza di questa parte del programma è, la sua immensa complessità. Sarà un lavoro lungo trascriverlo tutto e senza dubbio ci saranno molti errori in questa fase, ragion per cui dovrete affidarvi alle tavole di controllo per esaminare il risultato. Al termine di questo lavoro avrete lo stesso programma che abbiamo utilizzato noi per sviluppare tutte le routine in linguaggio macchina contenute nel testo. Il programma funziona e questo basta a giustificare lo sforzo che richiede per copiarlo.

SEZIONE 1: inizializzazione

MODULO 4.1

```
19046 TA$(2) = TA$(2)+"BYTWRDDBYENDORGPR
TSYM"
19047 T$ =    "61210690B0F02430D01000507
018D858"
19048 T$ = T$+"B8CDECCCCECA884DEEE8C84C2
0ADAEAC"
19049 T$ = T$+"4AEA0D480868282A6A4060ED3
8F8788D"
19050 T$ = T$+"8E8CAAA8BA8A9A98"
19051 GOSUB 12200 : TA$(3) = T1$
```

```
19052 T$ =     "FF11FFFFFF090AFFFF1D0EFFF
F051EFF"
19053 T$ = T$+"FF15FFFFFFFFFFFFFF01FFFFF
F1916FF"
19054 T$ = T$+"FF2DFFFF2C293EFFFF3D2EFFF
F2526FF"
19055 T$ = T$+"FF35FFFFFFFFFFFFFF31FFFFF
F3936FF"
19056 T$ = T$+"FF51FFFFFF495EFFFF5D4EFF6
C4546FF"
19057 T$ = T$+"FF55FFFFFFFFFFFFFF41FFFFF
F5956FF"
19058 T$ = T$+"FF6DFFFFFF697EFFFF7D6EFFF
F6566FF"
19059 GOSUB 12200 : TA$(4) = T1$
19060 T$ =     "FF75FFFFFFFFFFFFFF71FFFFF
F7976FF"
19061 T$ = T$+"FF91FFFF949D96FFFFFFFFFF8
48586FF"
19062 T$ = T$+"FF95FFFFFFFFFFFFFF81FFFFF
F99FFFF"
19063 T$ = T$+"BCB1BEFFA0A9A2FFFFBDFFAFA
4A5A6FF"
19064 T$ = T$+"FFB5FFFFFFFFFFFFFFA1FFFFB
4B9B6FF"
19065 T$ = T$+"FFD1FFFFC0C9DEFFFFDDFFFFC
4C5C6FF"
19066 T$ = T$+"FFD5FFFFFFFFFFFFFFC1FFFFF
FD9D6FF"
19067 GOSUB 12200 : TA$(4) = TA$(4)+T1$
19068 T$ =     "FFF1FFFFE0E9FEFFFFFDFFFFE
4E5E6FF"
19069 T$ = T$+"FFF5FFFFFFFFFFFFFFE1FFFFF
FF9F6"
19070 GOSUB 12200 : TA$(4) = TA$(4)+T1$
19080 SM = 50 : SE = 0 : DIM STABLE$(SM)
19101 DIM ERR$(18)
19103 ERR$(1) = "SINGLE BYTE OUT OF RANG
E"
19104 ERR$(2) = "DOUBLE BYTE OUT OF RANG
E"
19105 ERR$(3) = "INVALID OPRAND OR OPCOD
E"
19106 ERR$(4) = "INVALID OPERATOR"
19107 ERR$(5) = "INDEX IS NOT X OR Y"
19108 ERR$(6) = "LABEL NOT ALPHA-NUMERIC
"
19109 ERR$(7) = "INCORRECT NUMBER BASE"
```

```
19110 ERR$(8) = "LABEL DEFINED TWICE"
19112 ERR$(10) = "BRANCH OUT OF RANGE"
19113 ERR$(11) = "UNDEFINED LABEL"
19114 ERR$(12) = "ONLY SINGLE CHR. EXPEC
TED"
19116 ERR$(14) = "OUT OF SYMBOL SPACE"
19117 ERR$(15) = "DIVISION BY ZERO"
19120 ERR$(18) = "ADDRESSING MODE NOT AV
AILBLE WITH THIS OPCODE"
19980 DIM FI$(254) : GOSUB 24300
19990 RETURN
```

Se avete preso nota di ciò che avete finora trascritto, capirete immediatamente che il presente modulo non può restare completamente isolato, ma va congiunto ad un altro già esistente e precisamente al modulo di inizializzazione per le tabelle dei tipi di codici operativi e degli operandi sulle quali lavora il disassembler. Nel caso dell'assembler si useranno le stesse tabelle, ma nella direzione opposta. Invece di cercare un valore in memoria e poi controllare l'appropriato codice mnemonico ed il modo di indirizzamento, l'assembler esaminerà i file in ingresso attraverso il file editor e cercherà di costruire l'equivalente di ciascuna linea in linguaggio macchina oppure di rifiutare la linea come istruzione non lecita.

Se ci riflettete, questo lavoro richiede alcune informazioni in più poiché, invece di essere in grado di leggere un valore e quindi scegliere un formato basato sul codice operativo, l'assembler deve, dopo aver trovato un'istruzione come 'load' all'inizio di una linea come 'LDA $AAAA.X', essere in grado di esaminare tutti i possibili formati di un'istruzione 'load' per vedere se la presente istruzione è lecita o no. Per far ciò sono state aggiunte altre due tabelle a quelle già memorizzate nel programma. Fra le linee 19047 e 19050 è situata una tabella di numeri esadecimali a due caratteri corrispondenti a ciascuno dei possibili codici operativi memorizzati in TA$(2) per il disassembler. Essi mostrano per ogni codice operativo il tipo del primo operando che può essere usato. Le linee 19052-19067 consistono di ulteriori tipi di operandi per gruppi particolari di codici operativi.

Più avanti nel programma vedremo come ogni possibile operando viene confrontato con quello effettivamente presente nell'istruzione in linguaggio assemblatore contenuta in una riga di FI$. Per il momento basti sapere che rilevando un'istruzione che inizi con ADC (il primo codice operativo a tre caratteri di TA$(2)) l'assembler passerà a TA$(3). Scoprirà così che questa potrebbe essere un'istruzione contenente il codice operativo 61 (esadecimale) ed esaminerà il formato dell'istruzione in linguaggio assemblatore per vedere se soddisfa il formato richiesto dal codice e-

sadecimale 61, cioé ADC($50,X). Se il formato della linea inserita non è conforme a quello richiesto dal codice operativo, allora il valore 61 esadecimale (97 decimale) verrà usato per trovare il prossimo codice operativo possibile in TA$(4). Questo si troverà nella 98.ma coppia di caratteri di TA$(4) (la numerazione parte sempre da zero) dove si trova il codice operativo 6D, che segnala un'altra istruzione comprendente ADC, ma questa volta della forma ADC $AAA. Il valore 6D viene quindi usato per trovare il successivo codice nella tabella che produrrebbe un'istruzione iniziante con ADC.

Nel caso di ADC ci sono otto possibili codici operativi e se dopo averli confrontati tutti con il formato dell'istruzione effettivamente contenuta nella riga di FI$ nessuno di essi corrisponderà, l'ultimo possibile codice operativo conterrà il valore FF, che indica il termine della catena di possibili istruzioni ADC e quindi nessun codice operativo legale corrisponderà all'istruzione immessa. Se provate voi stessi a percorrere le tabelle con un qualsiasi codice operativo a tre lettere, prima di tutto trovandone la posizione in TA$, quindi trovando l'inizio della corrispondente catena di codici operativi in TA$(3) e seguendola in TA$(4) dovreste riuscire in breve a vedere cosa succede. L'unica vera aggiunta alle tabelle già previste dal disassembler è fatta dalla linea 19046. Essa apparentemente aggiunge sette nuovi tipi di codici operativi alla lista su cui lavorava il disassembler. Queste sono le direttive assembler, cioè sette istruzioni che non sono istruzioni vere e proprie del linguaggio assemblatore, ma piuttosto istruzioni all'assembler di comportarsi in un certo modo durante l'elaborazione di un programma in linguaggio assemblatore. Le sette direttive, BYT, WRD, DBY, END, ORG, PRT e SYM, verranno spiegate più avanti nel programma.

Da 19100 a 19120 troverete i vari messaggi di errore che l'assembler è in grado di generare quando incontra un'istruzione non valida o omissioni nel programma. Anche questi verranno spiegati più approfonditamente nel seguito.

TAVOLA DI CONTROLLO

```
19046 193     19047 171     19048 189
19049 252     19050 170     19051 244
19052 251     19053 248     19054 187
19055 1       19056 187     19057 8
19058 238     19059 245     19060 79
19061 198     19062 53      19063 216
19064 38      19065 13      19066 68
19067 221     19068 91      19069 192
```

```
19070 221      19080 27       19101 109
19103 53       19104 47       19105 77
19106 91       19107 192      19108 1
19109 152      19110 215      19112 253
19113 8        19114 184      19116 40
19117 122      19120 37       19980 101
19990 142
```

MODULO 4.2

```
20000 REM********************************
20001 REM GENERATE ASSEMBLY LISTING
20002 REM********************************
20005 SE = 0 : FMAX = LEN(PTR$) : SY = F
ALSE
20010 INPUT " ERROR ONLY LISTING (Y/N)
 :"; T$
20020 EO = LEFT$ (T$,1)="Y"
20025 INPUT " ASSEMBLE TO MEMORY (Y/N)
 :"; T$
20029 AM = LEFT$ (T$,1)="Y"
20030 AD = 0 : REM SET START ADDRESS
20040 FOR Q = 1 TO FMAX
20050 IN$ = FILE$(ASC (MID$ (PTR$,Q,1)))
 : O$ = ""
20060 GOSUB 26400
20070 IF EXIT THEN Q=FMAX+1
20080 NEXT Q
20085 T = FRE(X)
20090 AD = 0 : EC = 0 : PRINT "ADD.  DAT
A      SOURCE CODE"
20100 FOR Q = 1 TO FMAX
20110 IN$ = FILE$(ASC (MID$ (PTR$,Q,1)))
 : O$ = ""
20120 Q1 = AD
20130 GOSUB 27600
20140 IF ERR THEN 20250
20145 IF EO THEN 20222
20150 H = Q1 : GOSUB 11000
20160 O$ = H$
20180 Q2 = 3 : IF LEN(O$)<Q2 THEN Q2 = L
EN(O$)
20185 Q1$ ="" : IF O$="" THEN 20221
```

```
20190 FOR Q3 = 1 TO Q2
20200 H = ASC (MID$ (O$,Q3,1)) : GOSUB 1
1000
20210 IF LEN(H$)=1 THEN H$ = "0"+H$
20220 Q1$ = Q1$+H$ : NEXT Q3
20221 PRINT O$ SPC(6-LEN(O$)) Q1$ SPC(8-
LEN(Q1$)) ; : GOSUB 28100
20222 IF NOT AM OR O$="" THEN 20250
20225 FOR X = 1 TO LEN(O$) : POKE Q1+X-1
,ASC (MID$ (O$,X,1)) : NEXT
20250 IF EXIT THEN Q = FMAX+1 : REM LEAV
E LOOP
20260 NEXT Q
20270 PRINT : PRINT " TOTAL ERRORS IN FI
LE ---" EC : PRINT
20280 IF SY THEN GOSUB 26900
20290 IF PEEK(152) <> 0 THEN PRINT#2 : C
LOSE2 : GOTO 20300
20295 GET T$ : IF T$="" THEN 20295
20300 RETURN
```

ESEMPIO DI ERRORI: lista dei soli errori

```
add.   data       source code
                  50 LBL000 LDQ $A000
======================
   LABEL DEFINED TWICE ERROR
                  60 JSR ($300)
===================
   ADDRESSING MODE NOT AVAILBLE WITH THI
S OPCODE ERROR
                  70 LDA #LBL000/H
==============================
   DIVISION BY ZERO ERROR
                  80 LDX #LBL000-LBL000/256
@256
=======================================
   INVALID OPERATOR ERROR
                  140 BCC LBL000
===========================
   BRANCH OUT OF RANGE ERROR
                  150 JMP LBL001
===========================
   UNDEFINED LABEL ERROR
                  150 JMP LBL001
===========================
   UNDEFINED LABEL ERROR
                  160 LBL000 RTS
======================
   LABEL DEFINED TWICE ERROR

 TOTAL ERRORS IN FILE --- 8


H                  0
LBL000             C800
 total number of symbols --- 2
```

ESEMPIO DI ERRORI: lista dell'intero programma

```
add.  data      source code
0               10 PRT
0               20 SYM
0               30 ORG $C800
C800            40 H = 0
                50 LBL000 LDQ $A000
====================
   LABEL DEFINED TWICE ERROR
C800            50 LBL000 LDQ $A000
                60 JSR ($300)
===================
   ADDRESSING MODE NOT AVAILBLE WITH THI
S OPCODE ERROR
                70 LDA #LBL000/H
============================
   DIVISION BY ZERO ERROR
                80 LDX #LBL000-LBL000/256
@256
=====================================
   INVALID OPERATOR ERROR
C804  8543      90 STA &103
C806  8642      100 STX &102
C808  60        110 RTS
C809            120 ORG $CA00
CA00  18        130 CLC
                140 BCC LBL000
==========================
   BRANCH OUT OF RANGE ERROR
                150 JMP LBL001
==========================
   UNDEFINED LABEL ERROR
                150 JMP LBL001
==========================
   UNDEFINED LABEL ERROR
                160 LBL000 RTS
====================
   LABEL DEFINED TWICE ERROR
CA06            160 LBL000 RTS

 TOTAL ERRORS IN FILE --- 8


H                 0
LBL000            C800
 total number of symbols --- 2
```

Nelle sezioni precedenti dell'intero programma abbiamo adottato l'approccio di spiegare tutti i moduli necessari al funzionamento di un modulo di controllo prima di introdurre il modulo stesso. Fare questo nel caso dell'assembler provocherebbe una serie di pagine di spiegazioni prima di poter offrire un quadro di ciò che il programma effettivamente fa. La grande complessità dell'assembler ci ha imposto di adottare un approccio 'top-down' e cercare di passare da una semplice spiegazione del funzionamento del programma ad un esame dettagliato dell'intero listing aggiungendo poi i dettagli necessari. È per questo motivo che iniziamo il nostro commento sulla parte principale dell'assembler da questo modulo principale di controllo. Il modulo da solo è totalmente inutile, non compie quasi alcun lavoro se non quello di distribuire i compiti ad altre parti del programma. Nonostante ciò, commentarlo a questo punto ci aiuterà ad offrire una panoramica necessaria del funzionamento dell'assembler.

Commento

20010-20029: l'assembler è in grado di compilare un programma in linguaggio macchina in quattro modi differenti. Può offrire un listing completo della istruzione in linguaggio assemblatore insieme all'elenco di ogni errore presente oppure può evitare il listing e fornire i soli errori. Un esempio di entrambe queste possibilità è mostrato al termine del modulo (v. sopra). Può anche essere istruito per piazzare in memoria i programmi in codice macchina risultanti dalla compilazione oppure per eseguirli lasciando intatta la memoria. Se ad esempio desiderate inserire in memoria in una posizione prestabilita una routine in codice macchina fra un punto di inizio ed uno di fine determinati senza alterare la memoria esterna ad essi, fareste bene a richiedere prima un listing completo del programma senza che esso venga posto in memoria. Esso mostrerà esattamente dove il codice macchina assemblato sarebbe stato posto in memoria prima di compiere qualcosa di irrimediabile!

20030-20085: prima di iniziare a lavorare sul programma in linguaggio assemblatore la variabile AD viene posta uguale a zero, il che significa che l'indirizzo dal quale partirà eventualmente il programma in linguaggio macchina è zero. Durante il programma in linguaggio assemblatore questo punto di partenza verrà quasi in ogni caso riinizializzato a qualche altra posizione in memoria per mezzo della direttiva ORG (origine). L'assembler ora elaborerà il programma due volte in quelli che vengono chiamati due 'passi'. Questo ciclo richiama le sezioni del programma che esegue il 'Passo 1'. Durante il passo 1 ogni variabile (compreso un tipo particolare di variabile detto 'label' – etichetta –, che definisce la corretta destinazione di memoria per un'istruzione di salto) viene esaminata e posta, con i valori ad essa associati, in una tabella nota come 'tavola simboli' che verrà usata nel successivo assemblaggio del programma.

Ogni linea del programma in linguaggio assemblatore viene ottenuta in IN$ prima di eseguire il passo 1. Al ritorno dall'esecuzione del passo 1 su una qualsiasi linea, viene fatto un controllo della variabile EXIT, che è posta uguale a TRUE se viene incontrata la direttiva END durante l'esame del programma. A questo punto l'assemblaggio ha termine anche se non si è raggiunto il termine di FI$. Al termine del ciclo viene richiamata la funzione FRE, che assicura il compattamento e l'eliminazione degli spazi inutili, in modo da evitare pericoli di traboccamento dalla memoria.

20090-20300: questo è il ciclo che controlla il secondo passo attraverso il programma da assemblatore. La routine per il secondo passo è richiamata alla linea 20130. Durante il secondo passo il programma verrà effettivamente assemblato, con la traduzione di ogni istruzione valida nei byte necessari a rappresentarla in codice macchina come codice operativo ed operando, inclusa la traduzione delle variabili in valori e l'assegnamento dei valori alle etichette usate per i salti. Al ritorno dalla routine viene eseguito un test della variabile ERR per vedere se è stato rilevato un errore. In questo caso nessun'altra elaborazione è eseguita su quell'istruzione. Se si è richiesto il listing dei soli errori (EO posta uguale a TRUE) la routine di stampa viene omessa. Nelle linee 20150-20221 viene visualizzata in modo standardizzato l'informazione ottenuta dal secondo passo. Viene incluso l'indirizzo in cui una istruzione verrà inserita in memoria se il programma verrà effettivamente assemblato in memoria, la rappresentazione esadecimale degli 1, 2 o 3 byte che vi saranno posti e l'istruzione originale in linguaggio assemblatore. I byte della necessaria istruzione macchina sono contenuti in 0$ e nelle linee 20222-20225. Se AM risulta TRUE, questi byte vengono posti in memoria iniziando dalla locazione opportuna. Viene eseguito un test per vedere se si è incontrata la direttiva END e viene prelevata la successiva linea di programma in caso di esito negativo. Infine, se si è trovata la direttiva SYM durante il corso del programma viene posta a TRUE la variabile SY e verrà visualizzata la tavola simboli al termine del listing del programma assemblato.

TAVOLA DI CONTROLLO

```
20000 123     20001 180     20002 123
20005 3       20010 227     20020 195
20025 197     20029 189     20030 144
20040 37      20050 139     20060 169
20070 214     20080 243     20085 167
20090 63      20100 37      20110 139
20120 249     20130 172     20140 116
20145 30      20150 213     20160 211
```

```
20180 109     20185 40      20190 175
20200 5       20210 221     20220 244
20221 86      20222 58      20225 198
20250 6       20260 243     20270 10
20280 236     20290 63      20295 181
20300 142
```

SEZIONE 2: routine del passo 1

MODULO 4.3

```
26400 REM********************************
26401 REM DO PASS 1 ASSEMBLY ON IN$
26402 REM********************************
26405 PRINT "[HOM][CDW][CDW][CDW][CDW][C
DW][CDW][CDW][CDW][CDW][CDW][CDW][CDW][C
DW][CDW][CDW][CDW][CDW][CDW][CDW][CDW][C
DW][CDW]
        " ;
26406 PRINT "
                " ;
26407 PRINT "[HOM][CDW][CDW][CDW][CDW][C
DW][CDW][CDW][CDW][CDW][CDW][CDW][CDW][C
DW][CDW][CDW][CDW][CDW][CDW][CDW][CDW][C
DW][CDW]" ;: GOSUB 28100
26410 PASS = 1 : EXIT = FALSE : PTR = 2
26420 GOSUB 28850
26430 IF NOT ERR THEN 26540
26440 IF T=58 AND LEN(H$)=0 THEN 26420
26450 IF T=59 OR T=-1 THEN RETURN
26460 GOSUB 28700
26480 GOSUB 28850
26490 IF NOT ERR THEN 26540
26500 IF T=58 AND LEN(H$)=0 THEN 26420
26520 RETURN
26540 IF PO>55 THEN GOSUB 26600 : GOTO 2
6556
26550 GOSUB 26100
26552 GOSUB 26300 : IF ERR AND OP>3 AND
OP<7 THEN OP = OP+6 : GOTO 26552
26555 GOSUB 26560
26556 IF LEN(IN$)>PTR AND NOT EXIT THEN
26420
26557 RETURN
```

Avendo esaminato il modulo di controllo generale per i due passi, vediamo quello relativo al passo 1. Ancora una volta lasceremo al seguito le spiegazioni dettagliate dei singoli compiti e ci concentreremo su una descrizione del lavoro globale svolto durante questo passo.

Commento

26400-26407: queste linee cancellano le due righe inferiori dello schermo e vi stampano l'istruzione in linguaggio assemblatore correntemente elaborata, come traccia per l'utente del punto cui è giunta l'elaborazione del passo 1.

26410: la variabile PTR indica dove avrà inizio l'esame della linea corrente. È posta uguale a due per saltare i due byte contenenti il numero di linea.

26420: viene richiamata la routine che identifica il codice mnemonico usando la tabella in TA$(2). La routine esaminerà IN$ a partire dal carattere seguente quello indicato da PTR fino a trovare un carattere diverso da una lettera o da una cifra, quindi restituirà il controllo.

26430-26520: al ritorno dal precedente GOSUB, H$ conterrà una stringa di caratteri terminante con uno spazio, due punti o un qualsiasi altro carattere che non sia una cifra o una lettera. Se, al ritorno dal GOSUB precedente, non è stato trovato un codice operativo valido il cui simbolo mnemonico sia stato posto in H$, la linea 26440 controlla se è stato trovato qualche carattere prima dei due punti che separano le istruzioni sulla stessa linea (o un carattere qualsiasi che non sia una lettera o una cifra). In caso negativo, i due punti vengono ignorati e viene chiamata di nuovo la routine che ricerca un codice mnemonico, stavolta a partire dai due punti in avanti.

Nella linea 26540 viene eseguito un test per vedere se si è raggiunto un punto e virgola o la fine linea. In tal caso non viene intrapresa alcuna attività sulla linea corrente. In questo modo si possono inserire commenti nel programma senza confusione facendoli iniziare con un punto e virgola. Se H$ contiene dei caratteri allora, dal momento che non sono stati identificati come codici mnemonici, vengono intesi come etichetta ed il nome viene immesso nella tavola simboli dalla subroutine alla riga 28700. Una volta posta l'etichetta nella tavola simboli, il programma inizia la ricerca del codice operativo mnemonico che dovrebbe seguirla.

26540: a questo punto dev'essere stato identificato un codice mnemonico valido. Se occupa una posizione nelle tabelle, indicata dalla variabile PO, maggiore di 55 esso rappresenta una direttiva assembler e viene richiamata la subroutine nella linea 26600 per valutarla.

26550: a questo punto, trovato un codice mnemonico valido, viene richiamata la subroutine che valuta il tipo di operando.

26552: ora abbiamo un codice mnemonico lecito e, si spera, un operando di tipo lecito. Non abbiamo ancora la garanzia che il tipo dell'operando sia compatibile con quel codice operativo. Questa linea richiama la subroutine che stabilisce se essi formano una coppia effettivamente valida. La subroutine, nel caso incontri un indirizzo che potrebbe essere raggiunto con il modo di indirizzamento in pagina zero, supporrà che quel tipo di indirizzamento sia effettivamente quello usato, benché ciò possa provocare una segnalazione d'errore nel caso il codice operativo in esame non possa usare l'indirizzamento in pagina zero. Al ritorno dalla subroutine, se è stato selezionato l'indirizzamento in pagina zero ed è stato segnalato un errore, il tipo dell'operando, rappresentato dalla variabile OP, viene incrementato di 6 per trasformarlo in un modo di indirizzamento assoluto.

26555: nella valutazione di un'istruzione in linguaggio assemblatore c'è sempre la necessità di tener conto di quanti byte richiederà l'istruzione una volta assemblata, in modo che l'istruzione seguente inizi al punto giusto in memoria. Questo viene realizzato dalla subroutine alla riga 26560.

26556: se l'analisi della linea non è ancora terminata e non si è raggiunto un 'END', il resto della linea viene elaborato allo stesso modo.

TAVOLA DI CONTROLLO

```
26400 123     26401 2       26402 123
26405 225     26406 56      26407 227
26410 255     26420 180     26430 68
26440 84      26450 102     26460 174
26480 180     26490 68      26500 84
26520 142     26540 34      26550 166
26552 190     26555 176     26556 247
26557 142
```

MODULO 4.4

```
28100 REM******************************
28101 REM PRINT IN$ TO THE SCREEN
28102 REM******************************
28120 PRINT 256*ASC (MID$ (IN$,2,1))+ASC
 (MID$ (IN$,1,1)) MID$ (IN$,3)
28140 RETURN
```

Questo modulo si limita a stampare la linea corrente del programma in linguaggio assemblatore, compreso il suo numero di linea. Essa verrà stampata nella parte bassa dello schermo durante il passo 1. Nel passo 2 potrà essere inclusa nel programma una direttiva PRT per inviare l'uscita alla stampante.

TAVOLA DI CONTROLLO

```
28100 123     28101 187     28102 123
28120 80
```

MODULO 4.5

```
28150 REM******************************
28151 REM SYMBOL TO NON-LETTER/DIGIT
28152 REM******************************
28160 H$ = "" : T = -1
28165 PTR = PTR+1
28170 IF PTR>LEN(IN$) THEN 28210
28180 T = ASC (MID$ (IN$,PTR,1))
28185 IF T=32 AND LEN(H$)=0 THEN 28160
28190 IF T<48 OR T>90 OR ( T>57 AND T<65
 ) THEN 28210
28200 H$ = H$+CHR$ (T) : GOTO 28165
28210 RETURN
```

Questo semplice modulo esamina la linea in IN$ a partire dal punto indicato dalla variabile PTR, componendo una stringa, HS, da cui vengono eliminati gli spazi i-

niziali e che termina non appena si incontri un carattere che non sia una lettera o un numero. Il modulo viene usato dal successivo per restituire una stringa contenente un codice menmonico o un'etichetta.

TAVOLA DI CONTROLLO

```
28150 123     28151 240     28152 123
28160 62      28165 185     28170 5
28180 178     28185 79      28190 179
28200 9       28210 142
```

MODULO 4.6

```
28850 REM******************************
28851 REM TEST FOR OPCODE/DIRECTIVE
28852 REM******************************
28860 GOSUB 28150
28870 ERR = FALSE
28890 PTR = PTR-1
28895 IF LEN(H$)<>3 THEN 28940
28900 PO = -2
28910 PO = PO+3
28920 IF H$=MID$ (TA$(2),PO,3) THEN 2895
0
28930 IF (PO+3)<=LEN(TA$(2)) THEN 28910
28940 ERR = TRUE
28950 PO = (PO-1)/3
28960 ERR = (PO=56) OR ERR
28970 IF PO>56 THEN PO = PO-1
28980 RETURN
```

Una volta ottenuta una stringa di caratteri che può contenere o meno un codice operativo valido o un'etichetta, iniziamo il procedimento vero e proprio di valutazione di ciò che si è trovato. Il metodo di ricerca del codice operativo corretto è stato descritto nel modulo 1 ed il procedimento inizia con questo modulo.

Commento

28890: se la stringa restituita in H$ non è lunga tre caratteri, non può rappresentare un codice operativo valido e quindi non ha senso esaminare la tabella.

28895: PTR è riinizializzato per puntare all'ultimo carattere di H$ nella linea da cui è stato estratto.

28900-28940: questo ciclo esamina TA$(2) – che contiene i codici mnemonici leciti a tre lettere – per confrontare ogni gruppo di caratteri con quanto estratto da H$. Il puntatore per questo scopo è inizializzato a –2 di modo che alla prima iterazione del ciclo, quando viene aggiunto un tre, la ricerca parta dalla posizione uno. Se viene eseguito l'intero ciclo e si raggiunge la linea 28940, i tre caratteri esaminati non corrispondono ad alcun codice mnemonico lecito.

28950: l'indirizzo in TA$(2), che veniva incrementato con passo 3, è ora modificato in modo che, ad esempio, i tre caratteri nella posizione 19-21 siano ora identificati in PO come il codice mnemonico 7.

28960: questa strana linea deve tener conto del fatto che le tabelle del disassembler su cui lavora l'assemblatore contengono i tre caratteri '???' che vengono usati quando il disassembler non riesce ad associare un codice operativo valido al contenuto della memoria. Senza questo controllo si potrebbe immettere '???' in un programma in linguaggio assemblatore e creare così confusione al momento del suo riconoscimento come codice mnemonico valido. La linea segnala un errore se H$ consiste di '???'.

28970: anche questa linea deve tener conto di '???' nella tabella. Le direttive assembler cadono dopo il punto di domanda, quindi al momento di rinumerarle per gli scopi dell'assembler la loro posizione è diminuita di uno.

TAVOLA DI CONTROLLO

```
28850 123     28851 158     28852 123
28860 173     28870 70      28890 186
28895 176     28900 110     28910 13
28920 57      28930 24      28940 27
28950 61      28960 193     28970 89
28980 142
```

```
28700 REM********************************
28701 REM ADD SYMBOL TO SYMBOL TABLE
28702 REM********************************
28710 IF SE>=SM THEN EXIT = TRUE : PASS
= 2 : EN = 14 : GOTO 28000
28720 GOSUB 28250 : IF NOT ERR THEN 2883
0
28740 T$ = LEFT$ (H$+"        ",6)
28745 TB = PTR
28750 GOSUB 28150 : REM DOES = FOLLOW
28760 IF T<>61 THEN PTR = TB : RE = AD :
 GOTO 28780
28770 T0 = T : GOSUB 28600
28780 EN = 0
28790 IF RE<0 OR RE>65535 THEN ST$(SE)=T
$+CHR$ (0)+CHR$ (0)+CHR$ (2) : GOTO 2881
0
28800 ST$(SE) = T$+CHR$ (RE-INT(RE/256)*
256)+CHR$ (INT(RE/256))
28810 SE = SE+1
28820 RETURN
28830 IF PASS=1 AND LEN(ST$(T1))<9 THEN
ST$(T1) = ST$(T1)+CHR$ (8)
28835 IF PASS<>2 THEN 28840
28836 TA = PTR : GOSUB 28150 : IF T<>61
THEN PTR = TA : GOTO 28840
28837 GOSUB 26000 : REM SCAN PAST = SIGN
 (IF PRESENT) ON PASS 2
28840 IF PASS<>2 OR LEN(ST$(T1))<9 THEN
RETURN
28845 EN = ASC (MID$ (ST$(T1),9,1)) : GO
TO 28000
```

Questo è il modulo di controllo per la valutazione di variabili ed etichette e si occupa anche di inserirle nella tavola simboli se non sono state già definite in precedenza. Nel modulo 2 un'etichetta era stata descritta come un tipo di variabile per amore di semplicità. Di fatto un'etichetta è una costante che identifica un punto in un programma in linguaggio macchina cui un salto deve giungere. Usando le etichette diviene possibile assemblare un programma in linguaggio macchina in posti differenti della memoria senza dover ricalcolare gli indirizzi cui va fatto il salto. L'assembler individuerà automaticamente la posizione in memoria di un'istruzione etichettata e sostituirà il salto a quell'etichetta con un salto all'indirizzo opportuno.

28170: se ci sono più simboli di quanti la tavola simboli ne possa contenere (50) l'assemblaggio del programma cessa immediatamente e viene segnalato l'errore corrispondente.

28730: l'etichetta in H$ viene ora inviata al prossimo modulo, che esamina la tavola simboli (ST$) per vedere se è già presente in essa. Se lo è, ERR è restituita col valore 'falso' e viene segnalato un errore di 'label defined twice' (etichetta definita due volte). Questo uso invertito dell'errore può sembrare confusionario, ma è necessario dal momento che un uso eccessivo del modulo, alla linea 28250, richiederà la segnalazione di errore se un'etichetta non è presente nella tavola simboli.

28740: H$ viene ampliato alla lunghezza di sei caratteri nel caso sia più corto. Sei caratteri è la lunghezza massima per un'etichetta.

28745-28760: il contenuto di H$, se è valido, può essere una variabile oppure un'etichetta. Se è un'etichetta tutto ciò che l'assembler ha bisogno di sapere è l'indirizzo dell'istruzione così etichettata. Se è una variabile sarà seguita da un '=' per assegnarle un valore. Il modulo esaminatore viene richiamato alla linea 28150 per ricavare il carattere seguente. Se il primo carattere dopo la variabile (trascurando gli spazi) non è un segno di uguale, dev'essere un'etichetta e RE (risultato) viene usata per memorizzare l'indirizzo corrente della istruzione. PTR viene ripristinata al termine dell'etichetta usando la variabile temporanea TB.

28870: se il contenuto di H$ è una variabile, viene richiamato la sezione 'valutatore delle espressioni' del programma. Non tenteremo di spiegare il funzionamento di questo valutatore fino al termine del programma, dal momento che interromperebbe il nostro tentativo di seguire le principali attività dell'assembler. Per ora basta sapere che una chiamata al valutatore di espressioni nel caso di una linea come VAR=256 BYTE 1+15 restituirebbe nella variabile RE il risultato della parte destra dell'equazione. Alla fine tutto diverrà chiaro!

28790: se RE è minore di zero o maggiore di 65535 (massimo valore ottenibile che la CPU possa trattare in una sola istruzione) allora vengono aggiunti al nome della variabile o dell'etichetta nella tavola simboli due caratteri che rappresentano un risultato di 0 ed un flag che segnala un errore di 'doppio byte al di fuori dell'intervallo'.

28800: se RE è un numero valido allora viene aggiunto nella forma a due byte al

termine del nome della variabile o dell'etichetta in ST$. Dal momento che il nome è stato reso di lunghezza 6 in ogni caso, sarà facile recuperare il valore di RE per ogni variabile o etichetta.

28830: si arriva a questa linea solo se il nome corrente è già presente nella tavola simboli. Se non è seguito da un codice d'errore, viene aggiunto ora il codice d'errore 8, che sta per 'etichetta definita due volte'.

28835-28837: anche questo modulo viene usato nel passo 2. Nel passo 2 il nome è già presente in tavola simboli (piazzatovi durante il passo uno) e così la prima parte del modulo non verrà eseguita. Una volta ricavato il nome in H$, se il carattere successivo è un uguale, il risultato dell'espressione è già stato ottenuto ed il modulo alla linea 26000, che trova il termine di un'espressione o di una riga, viene richiamato per superare il resto dell'espressione.

28840: i messaggi d'errore sono stampati solo durante il passo 2 e, ovviamente, solo se si trova un codice d'errore al termine di un nome della tavola simboli.

TAVOLA DI CONTROLLO

```
28700 123     28701 175     28702 123
28710 239     28720 112     28740 31
28745 126     28750 81      28760 227
28770 241     28780 181     28790 40
28800 251     28810 253     28820 142
28830 106     28835 101     28836 117
28837 160     28840 126     28845 55
```

MODULO 4.8

```
28250 REM********************************
28251 REM FIND LABEL IN ST$
28252 REM********************************
28260 ERR = FALSE : H = 0 : T1 = 0
28270 IF LEN(H$)<6 THEN H$ = H$+" " : GO
TO 28270
```

```
28280 IF T1=SE THEN ERR = TRUE : RETURN
28290 IF MID$ (ST$(T1),1,6)<>H$ THEN T1
= T1+1 : GOTO 28280
28295 H = ASC (MID$ (ST$(T1),8,1))*256+A
SC (MID$ (ST$(T1),7,1)) : RETURN
```

Questo è un modulo richiamato dal precedente per determinare se un nome di variabile o un'etichetta è già presente nella tavola simboli. Il modulo ritorna un segnale d'errore di tipo 1 o 2 per indicare la presenza o l'assenza della variabile o etichetta. Viene anche restituito, se c'è, il contenuto dei caratteri 7 e 8.

TAVOLA DI CONTROLLO

```
28250 123     28251 242     28252 123
28260 75      28270 249     28280 132
28290 219     28295 92
```

MODULO 4.9

```
28000 REM********************************
28001 REM ASSEMBLER ERROR ROUTINE
28002 REM********************************
28005 IF PTR>=300 OR PASS<>2 THEN 28050
: REM SUPRESS SECONDARY ERRORS IN LINE
28010 PRINT SPC(14) ; : GOSUB 28100
28015 EC = EC+1
28020 FOR X = -13 TO PTR : PRINT "=" ;:
NEXT X : PRINT "[CUP]"
28030 PRINT "    " ERR$(EN) " ERROR"
28040 PTR = 300 : ERR = TRUE
28050 RETURN
```

Questo modulo, che viene chiamato dalla linea 28840 del modulo 4.7, stampa un messaggio se è stato segnalato un errore.

Commento

28005: se è già stato segnalato un errore per la linea corrente, si assegna alla variabile PTR il valore impossibile di 300 (la lunghezza massima di una stringa è 255) e non vengono stampati altri errori per quella linea. Gli errori non vengono stampati durante il passo 1.

28010: nel passo 2, l'indirizzo di memoria e i byte dati occupano 14 spazi in una linea. Quando si stampa una linea in cui dev'essere segnalato un errore, l'indirizzo e i dati non vengono stampati.

28020-28030: l'errore non è segnalato solo da un messaggio d'errore in uscita, viene anche stampato un puntatore alla posizione approssimativa in cui è stato rilevato l'errore nella linea, secondo quanto indicato dal valore di PTR.

TAVOLA DI CONTROLLO

```
28000 123     28001 61      28002 123
28005 92      28010 106     28015 221
28020 63      28030 124     28040 16
28050 142
```

MODULO 4.10

```
26000 REM*******************************
26001 REM SYMBOL UP TO COLON ETC.
26002 REM*******************************
26010 H$ = "" : T1 = LEN(IN$)
26020 PTR = PTR+1
26030 IF T1<P.TR THEN 26060
26040 T = ASC (MID$ (IN$,PTR,1))
26045 IF T=32 THEN 26020
26050 IF T<>58 AND T<>59 THEN H$ = H$+CH
R$ (T) : GOTO 26020
26060 RETURN
```

Questo modulo è simile al modulo nella linea 28150 (modulo 5) ma il suo scopo è differente, dovendo determinare il termine di un'istruzione in linguaggio assem-

blatore. Differisce dal modulo 5 in quanto non termina col ritrovamento di un carattere diverso da una cifra o una lettera, ma solo se trova un delimitatore come due punti, punto e virgola o il termine della riga, tralasciando tutti gli spazi presenti nella linea originale.

TAVOLA DI CONTROLLO

```
26000 123     26001 210     26002 123
26010 98      26020 185     26030 190
26040 178     26045 247     26050 201
26060 142
```

MODULO 4.11

```
26600 REM*******************************
26601 REM CALCULATE DIRECTIVE LENGTH
26602 REM*******************************
26610 T1 = LEN(IN$)
26620 IF PO=56 THEN 26720 : REM BYT DIRE
CTIVE
26625 IF PO=60 THEN GOSUB 28600 : AD = R
ESULT : REM DEAL WITH ORG DIRECTIVE
26627 IF PO=59 THEN EXIT = TRUE
26630 IF PO>58 THEN RETURN : REM END & O
RG DIRECTIVES
26640 REM FIND LEN. OF WRD & DBY
26650 AD = AD+2
26660 PTR = PTR+1
26670 IF PTR>T1 THEN RETURN
26680 T = ASC (MID$ (IN$,PTR,1))
26690 IF T=58 OR T=59 THEN RETURN
26700 IF T<>46 THEN 26660
26710 GOTO 26650
26720 REM LENGTH FOR BYT.
26730 AD = AD+1
26740 PTR = PTR+1
26750 IF PTR>T1 THEN RETURN
26760 T = ASC (MID$ (IN$,PTR,1))
26770 IF T=58 OR T=59 THEN RETURN
26780 IF T<>46 THEN 26740
26790 GOTO 26730
```

Questo modulo viene usato esclusivamente nel passo 1 ed agisce su quelle direttive assembler che sono rilevanti ai fini di quella parte dell'esecuzione del programma e cioè: BYT, WRD, DBY, END, ORG. Il modulo viene richiamato dal modulo 3 ogni volta che venga individuata una direttiva assembler.

Commento

26620: si occupa della direttiva BYT che verrà discussa alla linea 26720.

26625: 60 è il codice per la direttiva ORG che viene usata per stabilire l'indirizzo in memoria su cui si baserà il successivo assemblaggio. ORG verrà seguito sulla linea da un'espressione ed il 'valutatore di espressioni' (non ancora spiegato) viene richiamato per ottenere dall'espressione l'indirizzo desiderato. AD, l'indirizzo cui verrà inserita l'istruzione macchina successiva, è quindi posto uguale al valore dell'espressione.

26627: se si incontra la direttiva END si pone a TRUE la variabile EXIT.

26630: a differenza di ORG ed END, nessuna delle direttive con codice superiore a 58 (SYM, PRT) interessa l'esecuzione del programma durante il passo 1.

26640-26710: a questo punto dell'esecuzione del programma, deve essere stata incontrata una delle due direttive WRD e DBY. Nel programma in linguaggio assemblatore esse prendono la forma: WRD (o DBY) $AAAA.$BBBB.$CCCC, cioè la direttiva è seguita da una serie di valori a due bytes che verranno immessi direttamente in memoria – permettendo così di definire una tabella.

La differenza fra le due direttive è che WRD prenderà i due byte specificati ad esempio con $ABCD e li memorizzerà nell'ordine CD,AB mentre DBY lo farà nell'ordine AB,CD. La CPU lavora solitamente su numeri a due byte in cui il byte meno significativo (nel nostro caso CD) viene per primo. Il problema con queste due direttive nel passo 1 è che, come abbiamo già notato, si deve tener conto della lunghezza in byte di ciascuna istruzione in modo che si possa dare alle etichette, quando vengono definite nel passo 1, il loro corretto indirizzo in memoria. BYT e WRD possono essere seguiti da un numero qualsiasi di valori, fino ad una stringa di lunghezza massima 255. Questo ciclo perciò esamina la linea, contando il numero di valori a due byte ed incrementando il contatore di indirizzo AD di due per ogni valore trovato.

26720-26790: l'istruzione BYT specifica valori ad un solo byte da memorizzare.

Questo ciclo svolge una funzione simile a quella del precedente, incrementando però il contatore di indirizzo di uno per ogni valore specificato.

TAVOLA DI CONTROLLO

```
26600 123     26601 222     26602 123
26610 70      26620 38      26625 40
26627 189     26630 61      26640 4
26650 216     26660 185     26670 76
26680 178     26690 247     26700 184
26710 172     26720 181     26730 215
26740 185     26750 76      26760 178
26770 247     26780 183     26790 171
```

MODULO 4.12

```
26100 REM********************************
26101 REM OPERAND TYPE TO BE USED
26102 REM********************************
26110 T6 = PTR : GOSUB 26000
26120 ERR = FALSE
26130 IF LEN(H$)=0 THEN OP = 1 : RETURN
26140 IF H$="A" THEN OP = 0 : RETURN
26145 IF ASC (H$)=35 THEN OP = 2 : RETUR
N
26170 OP = 12
26180 IF LEFT$ (H$,1)="(" THEN OP = OP-3
26190 T = 1 : T1 = LEN(H$)
26200 T2 = ASC (MID$ (H$,T,1))
26210 IF T2<>46 AND T<T1 THEN T = T+1 :
GOTO 26200
26220 IF T2<>46 THEN 26275
26230 T = T+1 : IF T>T1 THEN 26270
26240 T2 = ASC (MID$ (H$,T,1))
26250 IF T2=89 THEN OP = OP-1 : GOTO 262
75
26260 IF T2=88 THEN OP = OP-2 : GOTO 262
75
26270 REM NOT A VALID INDEX
26272 EN = 5 : GOTO 28000
26275 IF(OP=12)AND((PO>2ANDPO<6)OR(PO>6A
NDPO<10)ORPO=12ORPO=11)THEN OP = 3
```

```
26281 REM ZERO PAGE OPRANDS
26282 IF OP<10 THEN RETURN
26284 T7 = PTR : PTR = T6
26286 GOSUB 28600
26288 IF ERR OR RESULT>255 THEN 26292
26290 OP = OP-6
26292 PTR = T7
26294 RETURN
```

Seguendo lo sviluppo del passo 1 tracciato dal modulo di controllo, incontriamo ora le due routine che determinano il tipo di operando da usare e se quel tipo di operando è adatto al codice operativo, ricordando che non tutti i modi di indirizzamento possono venir usati con ogni codice operativo. Lo scopo di questo modulo è di determinare il tipo di operando che si addice al formato mostrato nell'istruzione.

Commento

26110: a questo punto PTR indica il carattere seguente il codice mnemonico e viene richiamata la subroutine in linea 26000 per ottenere la parte operando dell'istruzione, privata degli spazi.

26130: se l'operando ha lunghezza zero, il modo d'indirizzamento è quello implicito ed il valore di OP è zero.

26140: se l'operando è 'A' allora si tratta di indirizzamento in accumulatore, OP=1.

26145: se il primo carattere dell'operando è ' ', si tratta di indirizzamento immediato, OP=2.

26150-26160: si fa ora l'ipotesi temporanea che il tipo di operando sia 3, modo di indirizzamento relativo. La subroutine seguente viene richiamata per vedere se le tabelle in TA$ indicano che ciò è possibile per il codice operativo ricavato (cioè il codice operativo deve rappresentare un 'branch' di qualche genere). Il metodo per far ciò viene descritto nel prossimo modulo. I riferimenti a O$ interessano solo per il passo 2. La subroutine in linea 26300 inserirà in O$ il byte il cui valore rappresenta il codice operativo determinato.

A questo punto invece usiamo la subroutine solo per determinare se il tipo dell'operando è 3 — non vogliamo incrementare O$ a questo punto, così prendiamo nota della sua lunghezza prima che venga richiamata la subroutine e quindi lo riinizializziamo alla stessa lunghezza. Se viene segnalato un errore al ritorno dalla subroutine in linea 26300 allora l'operando di tipo 3 non è adatto al codice operativo che abbiamo e la subroutine procede.

26170-26260: queste linee sono un'immagine speculare di quelle del disassembler che determinano il formato dell'operando dal valore del byte operando di un'istruzione macchina. In questo caso ricaviamo il tipo di operando dal formato dell'istruzione.

26270-26280: se si rileva un punto ed il formato non corrisponde a quello dell'indirizzamento indicizzato viene segnalato un errore di tipo 5 — l'indice non è X o Y.

26282-26294: queste linee controllano se è possibile eseguire l'istruzione con il modo di indirizzamento in pagina zero — il formato per l'indirizzamento assoluto e quello per la pagina zero sono uguali e si è fatta sino a questo punto l'ipotesi che gli operandi che potevano essere entrambe le cose fossero indirizzamenti assoluti. Questo è possibile solo con tipi di operando maggiore di 10, che rappresentano i modi di di indirizzamento assoluti. PTR è riportato al valore iniziale al termine del codice operativo e l'operando viene rivalutato dal valutatore di espressioni. Se il risultato è nell'intervallo 0-255 allora è possibile usare il più veloce modo di indirizzamento in pagina zero ed il tipo dell'operando è ridotto di 6 per trasformare il modo di indirizzamento in un qualche tipo di indirizzamento in pagina zero.

TAVOLA DI CONTROLLO

```
26100 123     26101 213     26102 123
26110 145     26120 70      26130 190
26140 254     26145 250     26170 244
26180 160     26190 232     26200 243
26210 243     26220 236     26230 12
26240 243     26250 112     26260 112
26270 41      26272 215     26275 213
26281 99      26282 211     26284 95
26286 173     26288 156     26290 17
26292 115     26294 142
```

```
26300 REM**********************************
26301 REM EVALUATE OPCODE
26302 REM**********************************
26310 T1 = 3 : T = PO
26320 T = ASC (MID$ (TA$(T1),T+1,1))
26330 IF T=255 THEN ERR = TRUE : RETURN
26340 T1 = 4 : T2 = ASC (MID$ (TA$(1),IN
T(T/2+1),1))
26350 IF (1 AND T)=0 THEN T2 = INT(T2/16
)
26355 T2 = T2 AND 15
26360 IF T2<>OP THEN 26320
26370 O$ = O$+CHR$ (T)
26380 ERR = FALSE
26390 RETURN
```

A questo punto facciamo di nuovo uso delle nuove tabelle aggiunte a TA$ nel primo modulo dell'assembler. Lo scopo del modulo è quello di confrontare il codice operativo ottenuto con il tipo di operando per vedere se effettivamente sono compatibili. In caso contrario deve venir segnalato un errore.

Commento

26230: T è stato posto uguale a PO, la posizione del codice mnemonico in TA$(2). Così l'equazione fra stringhe in questa linea punta ad una coppia di byte in TA$(3). TA$(3) contiene, per ogni possibile tipo di codice operativo, la prima delle forme di byte possibili che quel codice operativo può assumere. Il valore è anch'esso il primo elemento nella catena delle possibili forme di byte del codice operativo.

26330: se, con ripetute iterazioni, il valore trovato nella tabella (successivamente TA$(4)) è FF esadecimale, non ci sono altre forme di codice operativo disponibili e viene segnalato un errore.

26340-26355: una volta trovato il codice operativo possibile, esso viene confrontato col modo d'indirizzamento necessario in TA$(1). I modi d'indirizzamento per ciascun codice operativo sono memorizzati in TA$(1). Un singolo carattere può venir usato per memorizzare due numeri fra 0 e 15 semplicemente moltiplicando uno dei numeri per 16 e poi sommandoli insieme. In questo modo, il modo d'indi-

rizzamento per il codice operativo uno nella tabella dei codici operativi si, troverà nel primo carattere di TA$(1), come quello per il codice operativo due.

Le linee 26350 e 26355 estraggono la parte necessaria del valore del carattere (0-15 e 16-255). Se la posizione del codice operativo (PO) nella tabella è dispari, allora viene usata la metà 'alta' del byte (T2/16) mentre se PO è pari viene usata la metà 'bassa' del byte (T2 AND 15).

26360: se il modo d'indirizzamento risultante non è lo stesso ottenuto esaminando l'operando dell'istruzione in linguaggio assemblatore, la subroutine torna alla linea 26320 e preleva la prossima forma possibile di codice operativo, insieme al modo d'indirizzamento ad essa associato e così via finché non ci siano altre forme di codici mnemonici.

26370: se l'esecuzione del programma ha raggiunto questo punto è perché è stato trovato nelle tabelle un modo d'indirizzamento adatto al formato dell'operando prelevato dall'istruzione in linguaggio assemblatore. Il codice operativo corretto per l'operando ed il tipo del codice operativo vengono aggiunti ad O$, che viene usato per memorizzare ciò che verrà poi immesso in memoria, sebbene questo abbia importanza solo nel passo 2.

TAVOLA DI CONTROLLO

```
26300 123     26301 224     26302 123
26310 9       26320 191     26330 87
26340 167     26350 81      26355 83
26360 24      26370 238     26380 70
26390 142
```

MODULO 4.14

```
26560 REM********************************
26561 REM BYTE LENGTH
26562 REM********************************
26565 AD = AD+1
26570 IF OP>1 THEN AD = AD+1
26580 IF OP>8 THEN AD = AD+1
26590 RETURN
```

Terminiamo l'esame del lavoro dell'assembler durante il passo 1 con questo breve modulo. Esso usa semplicemente il tipo di codice operativo per determinare quanti byte richiederà l'istruzione assemblata al momento di essere memorizzato nel passo 2. Questo affinché la variabile AD possa essere aggiornata correttamente allo scopo di definire le etichette.

TAVOLA DI CONTROLLO

```
26560 123     26561 197     26562 123
26565 215     26570 234     26580 241
26590 142
```

SEZIONE 3: routine del passo 2

MODULO 4.15

```
27600 REM*********************************
27601 REM DO PASS 2 ASSEMBLY
27602 REM*********************************
27605 PASS = 2
27610 O$ = ""
27620 EXIT = FALSE : ERR = FALSE
27625 PTR = 2
27630 GOSUB 28850
27640 IF NOT ERR THEN 27720
27650 IF T=58 AND LEN(H$)=0 THEN 27630
27660 IF T=59 OR T=-1 THEN ERR = FALSE
 RETURN
27665 GOSUB 28700
27670 GOSUB 28850
27680 IF NOT ERR THEN 27720
27690 IF T=58 AND LEN(H$)=0 THEN 27630
27695 IF T=59 OR T=-1 THEN ERR = FALSE
 RETURN
```

```
27700 EN = 3 : GOTO 28000
27720 IF PO>55 THEN GOSUB 27200 : GOTO 2
7745
27723 T5 = PTR : GOSUB 26100 : T8 = PTR
 : PTR = T5
27725 GOSUB 26300 : IF NOT ERR THEN 2773
0
27727 IF OP<7 AND OP>3 THEN OP = OP+6 :
PTR = T5 : GOTO 27725
27728 EN = 18 : GOTO 28000
27729 REM THIS BIT ATTEMPS TO MATCH ABSL
OUTE ADD MODE TO OPCODE IF ZP HAS FAILED
27730 GOSUB 26560
27740 IF NOT ERR AND LEN(O$)>0 THEN GOSU
B 27000 : PTR = T8
27745 IF LEN(IN$)>PTR AND NOT EXIT THEN
27630
27750 RETURN
```

Passiamo ora a fissare l'attenzione sul passo 2. Questo è il modulo di controllo per il passo 2 e, come abbiamo fatto per il modulo 4.3, seguiremo lo svolgimento generale dell'attività prima di passare ai dettagli.

Commento

27605-27700: ad eccezione dello svuotamento della stringa d'uscita (O$) e del fatto che la variabile PASS viene posta uguale a due, queste linee somigliano alla prima parte del modulo del passo 1. Viene fatto un test sul codice operativo e se il risultato è negativo si suppone che la prima parte della linea sia un'etichetta o una

variabile. Di conseguenza viene eseguita un'altra ricerca di codice operativo e se la linea non attribuisce un valore ad una variabile e non c'è alcun codice operativo presente viene segnalato un errore di tipo 3 'operando o codice operativo non valido'.

27720: se il tipo di codice operativo è maggiore di 55, è stata incontrata una direttiva assembler e viene richiamato il modulo opportuno per valutarla.

27723: viene richiamata la subroutine alla linea 26100 per valutare l'operando.

27725-27728: la subroutine alla linea 26300 esamina il confronto finora eseguito fra codice operativo e modo d'indirizzamento; tenta quindi di confrontarli nel modo d'indirizzamento assoluto, segnalando l'errore di tipo 18, 'modo d'indirizzamento non lecito per questo codice operativo', se il confronto non è positivo.

27730: con questa chiamata viene incrementato il contatore di byte AD.

27740: se non è stato trovato alcun errore e c'è qualcosa in O$, viene valutata la parte operando dell'istruzione. T8 è una variabile usata per muovere il puntatore oltre un indice del tipo 'X' alla fine dell'operando, dal momento che il valutatore dell'operando non esaminerà oltre. T8 è stata inizializzata nella linea 27723 dopo l'esecuzione della subroutine che valuta il tipo dell'operando ed esamina l'operando fino in fondo, incluso ogni indice seguente.

27745: questa linea permette la valutazione di più frasi sulla stessa linea di programma.

TAVOLA DI CONTROLLO

```
27600 123      27601 107      27602 123
27605 91       27610 169      27620 87
27625 26       27630 180      27640 69
27650 88       27660 38       27665 174
27670 180      27680 69       27690 88
27695 38       27700 213      27720 32
27723 138      27725 104      27727 178
27728 11       27729 248      27730 176
27740 46       27745 251      27750 142
```

```
27200 REM********************************
27201 REM DIRECTIVE OPERAND EVALUATOR
27202 REM********************************
27205 ERR = FALSE
27210 IF PO=60 THEN GOSUB 28600 : AD =RE
SULT
27214 IF PO=62 THEN SY = TRUE
27215 IF PO=61 THEN OPEN 2,4 : CMD 2 : P
RINT "[CDW]ADD.  DATA     SOURCE CODE[CD
W]"
27220 IF PO=59 THEN EXIT = TRUE
27230 IF PO>58 THEN RETURN
27240 IF PO=56 THEN 27330
27250 REM DBY & WRD DIRECTIVES
27260 GOSUB 28600
27270 IF RESULT<0 OR RESULT>65535 THEN E
N = 2 : GOTO 28000
27280 IF PO=58 THEN RESULT = INT(RESULT/
256)+256*(RESULT-INT(RESULT/256)*256)
27281 REM 27280 RESERVES HI. & LO. BYTES
 IF DIRECTIVE IS DBY
27290 T1 =T : GOSUB 27100 : AD = AD+2
27300 IF T1=32 THEN GOSUB 28150
27310 IF T1=46 THEN 27260
27320 RETURN
27330 REM BYT DIRECTIVE
27340 GOSUB 28600
27350 IF RESULT<0 OR RESULT>255 THEN EN
= 1 : GOTO 28000
27360 GOSUB 27140 : AD = AD+1
27370 IF T=32 THEN GOSUB 28150
27380 IF T=46 THEN 27340
27390 RETURN
```

Nel passo 1 abbiamo esaminato un modulo che determinava le azioni necessarie una volta incontrata una direttiva assembler e calcolava anche la lunghezza della stessa se si trattava di BYT, WRD o DBY. Questo modulo è simile, con la differenza che esegue azioni del tipo dell'apertura di un canale verso la stampante per l'u-

scita, della segnalazione SY di stampare la tavola simboli o del trasferimento di dati da una direttiva BYT, WRD o DBY alla stringa d'uscita O$.

Commento

27210: se si incontra una direttiva ORG, viene valutato il suo operando e il contatore di byte AD viene posto uguale al risultato.

27214: la presenza di SYM nel programma segnala a SY di stampare la tavola simboli al termine del listing del codice sorgente.

27215: la presenza di PRT nel programma apre un canale d'uscita verso la stampante per il listing del codice sorgente, altrimenti l'uscita è verso lo schermo.

27220: END provoca a questo punto il termine del passo.

27250-27320: il valore di una direttiva a due byte (DBY, WRD) è ottenuto usando il valutatore di espressioni ed i due byte vengono scambiati se la direttiva è DBY. Questo perché la routine alla linea 27100, che viene ora richiamata, piazza i due byte nella stringa O$ nell'ordine basso/alto. ad viene incrementato di due. La linea 27300 esamina il resto tralasciando gli spazi iniziali e la subroutine esegue un ciclo a ritroso per prelevare un altro doppio byte se si incontra un punto.

27340-27380: è una routine analoga a quella sopra descritta, ma per la direttiva a byte singolo BYT.

TAVOLA DI CONTROLLO

```
27200 123     27201 74      27202 123
27205 70      27210 166     27214 41
27215 154     27220 189     27230 221
27240 77      27250 243     27260 173
27270 176     27280 150     27281 31
27290 30      27300 219     27310 52
27320 142     27330 93      27340 173
27350 67      27360 252     27370 170
27380 2       27390 142
```

```
27000 REM********************************
27001 REM EVALUATE OPERAND
27002 REM********************************
27010 ERR = FALSE
27020 IF OP<2 THEN RETURN
27030 IF OP=3 THEN 27500
27040 IF OP=2 THEN 27400
27050 GOSUB 28600
27060 IF ERR OR LEN(O$)=0 THEN RETURN
27070 IF (RESULT<0 OR RESULT>255) AND OP
<9 THEN EN = 1 : GOTO 28000
27080 IF RESULT<0 OR RESULT>65535 THEN E
N= 2 : GOTO 28000
27090 IF OP<9 THEN 27140
27100 T = INT(RESULT/256)
27110 RESULT = RESULT-T*256
27120 GOSUB 27140
27130 RESULT = T
27140 O$ = O$+CHR$ (RESULT)
27150 RETURN
```

Questo modulo valuta un operando il cui tipo è già stato determinato, ponendo il risultato nella forma a uno o due byte in O$.

Commento

27020: se OP è minore di due non c'è operando.

27030: se OP è tre, è richiesto il modo d'indirizzamento relativo e viene richiamata la routine alla linea 27400.

27050-27080: per tutti gli altri valori di OP viene usato il valutatore di espressioni per ottenere un risultato, che verrà confrontato con i requisiti del modo di indirizzamento per uno o due byte.

27090-27140: queste sono due routine che pongono il risultato ottenuto dal valutatore di espressioni sotto forma di byte in O$. Notate che i numeri a due byte vengono posti in O$ con il byte 'alto' in seconda posizione.

TAVOLA DI CONTROLLO

```
27000 123     27001 47      27002 123
27010 70      27020 164     27030 20
27040 18      27050 173     27060 98
27070 14      27080 144     27090 27
27100 117     27110 248     27120 171
27130 37      27140 121     27150 142
```

MODULO 4.18

```
27500 REM********************************
27501 REM EVALUATE RELATIVE EXPRESSION
27502 REM********************************
27510 GOSUB 28600
27520 IF LEN(O$)=0 OR ERR THEN RETURN
27530 RESULT = RESULT-AD
27540 IF RESULT<0 THEN RESULT = RESULT+2
56
27560 IF RESULT<256 AND RESULT>=0 THEN 2
7140
27570 EN = 10
27580 GOTO 28000
```

Questo modulo valuta l'operando di un'istruzione usando l'indirizzamento relativo, in cui cioè un salto è specificato ddd dall'indirizzo corrente fino a 127 posizioni in più o a 128 in meno in memoria.

Commento

27530: dal momento che stiamo parlando di indirizzamento relativo, un salto è specificato in primo luogo all'indirizzo cui è diretto, quindi il salto relativo è calcolato sottraendo l'indirizzo corrente registrato in AD.

27540: salti negativi non possono venir immessi direttamente nel programma in linguaggio macchina, ma devono essere trasformati nella forma nota come 'complemento a due', in cui ogni numero negativo viene sommato a 256, in modo, da produrre un numero positivo nell'intervallo 128-255. Così i salti con valori superiori a 128 sono in realtà negativi ed il loro valore si ottiene sottraendo 256.

```
27500 123     27501 178     27502 123
27510 173     27520 98      27530 224
27540 75      27560 32      27570 230
27580 163
```

MODULO 4.19

```
27400 REM********************************
27401 REM EVALUATE IMMEDIATE EXPRESSION
27402 REM********************************
27410 T5 = PTR : GOSUB 26000
27420 IF ASC (H$)<>35 THEN 27480
27430 IF MID$ (H$,2,1)="'" THEN 27450
27440 PTR = T5
27442 IF PTR>LEN(IN$) THEN 27446
27444 IF ASC (MID$ (IN$,PTR,1))<>35 THEN
 PTR = PTR+1 : GOTO 27442
27446 OP = 8 : GOSUB 27050 : OP = 2
27448 RETURN
27450 REM SINGLE CHR. EXPECTED
27460 IF LEN(H$)<>3 THEN 27480
27470 O$ = O$+MID$ (H$,3,1) : RETURN
27480 EN = 12
27490 GOTO 28000
```

Questo modulo ricava il valore di un operando per un'istruzione concernente indirizzamenti immediati, cioé in cui viene posto direttamente in un registro un valore dell'intervallo 0-255.

Commento

27410-27420: l'operando viene ricavato in **H$** usando la routine alla linea 26000. Se non inizia con un ‘ ’ verrà segnalato un errore.

27430: se il secondo carattere dell'operando è un apice singolo, la routine si aspetterà un unico carattere, dopo l'apice, il cui valore ASCII sarà quello dell'operando. Non dovrebbe venir usato un secondo apice.

27442-27444: queste due linee trascurano ogni spazio in **IN$** fino al simbolo ‘hash’ (#).

27446: poiché l'indirizzamento immediato coinvolge un singolo byte, si usa la routine alla linea 27050 per valutare l'operando come se fosse il modo d'indirizzamento 8 (indiretto Y) e per porre il byte in O$. È semplicemente una scorciatoia.

27460-27470: queste due linee si occupano di caratteri singoli fra apici. Se la lunghezza di H$ non è tre allora il formato non è lecito e viene segnalato un errore. Se H$ è lungo tre caratteri, il carattere centrale viene preso come quello il cui valore ASCII rappresenta l'operando.

TAVOLA DI CONTROLLO

```
27400 123     27401 229     27402 123
27410 144     27420 230     27430 243
27440 113     27442 15      27444 169
27446 43      27448 142     27450 14
27460 174     27470 205     27480 232
27490 163
```

MODULO 4.20

```
26900 REM******************************
26901 REM DUMP SYMBOL TABLE TO SCREEN
26902 REM******************************
26910 IF SE<1 THEN 26975
26915 PRINT
26920 FOR X = 0 TO SE-1
26930 PRINT LEFT$ (ST$(X),6) TAB(10) ;
26940 H = ASC (MID$ (ST$(X),8))*256+ASC
(MID$ (ST$(X),7))
26950 GOSUB 11000
26960 PRINT.H$
26970 NEXT X
26975 PRINT "[CDW] TOTAL NUMBER OF SYMBO
LS ---" SE
26980 RETURN
```

Questo modulo non fa realmente parte del passo 2: semplicemente porta in uscita la tavola simboli al termine del passo 2 se la direttiva SYM era presente nel programma in linguaggio assemblatore.

```
26900 123     26901 6       26902 123
26910 27      26915 153     26920 115
26930 80      26940 64      26950 159
26960 37      26970 250     26975 248
26980 142
```

SEZIONE 4: il valutatore delle espressioni

Finora ci siamo riferiti molte volte alla misteriosa entità nota come 'valutatore di espressioni', prendendo per buono che facesse ciò che ci si attendeva da lui. Naturalmente sarebbe stato possibile scrivere un assembler che richiedesse che tutti i valori venissero espressi in decimale o esadecimale, ma si risparmia una grande quantità di tempo e di sforzi se l'utente è in grado di immettere variabili, saltare ad una posizione ad esempio sei byte più avanti di una particolare etichetta oppure calcolare byte immettendo qualcosa del tipo LDA VAL/256. Il valutatore di espressioni rende possibile tutto ciò, come scoprirete quando vi appresterete a trascrivere le nostre routine in linguaggio macchina.

MODULO 4.21

```
28300 REM*******************************
28301 REM EVALUATE LABEL OR NUMBER
28302 REM*******************************
28320 GOSUB 28150
28325 IF T=40 AND LEN(H$)=0 THEN GOSUB 2
8150
28330 T1 = LEN(H$)
28335 IF (T=-1 OR T=32 OR T=58 OR T=59 O
R T=41 OR T=46) AND T1 = 0 THEN RETURN
28340 IF T1=0 THEN 28390
28350 IF ASC (H$)<=57 THEN H = VAL(H$) :
 GOTO 28492
28360 GOSUB 28250 : REM FIND LABEL IN SY
MBOL TABLE
28370 IF ERR THEN EN = 11 : H = 0 : GOTO
 28000
28380 GOTO 28492
```

```
28390 REM HEX,OCTAL OR BINARY NUMBERS EV
ALUATE
28400 T2 = T : GOSUB 28150
28410 IF LEN(H$)=0 THEN 28450
28420 IF T2=36 THEN 28470
28430 IF T2=37 THEN BASE = 2 : GOTO 2847
0
28440 IF T2=38 THEN BASE = 8 : GOTO 2847
0
28450 REM INVALID LABEL
28460 H = 0 : EN = 6 : GOTO 28000
28470 REM TEST IF VALID NUMBER
28475 GOSUB 11950
28480 BASE = 16 : REM DEFAULT BASE
28490 IF ERR THEN H = 0 : EN = 7 : GOTO
28000
28492 PTR = PTR-1 : GOSUB 28150 : REM GE
T NEXT OPERATOR
28495 RETURN
```

TAVOLA DI CONTROLLO

```
28300 123     28301 48      28302 123
28320 173     28325 250     28330 247
28335 198     28340 255     28350 206
28360 173     28370 99      28380 178
28390 140     28400 243     28410 247
28420 56      28430 155     28440 162
28450 54      28460 188     28470 23
28475 173     28480 221     28490 56
28492 213     28495 142
```

Questo modulo è il nocciolo del valutatore di espressioni, dal momento che ha il compito di estrarre i valori di numeri o etichette, essendo permessi numeri in forma decimale, esadecimale ottale (base 8) o binaria (base 2). La subroutine restituisce un valore nella variabile H il cui intervallo possibile di valori è 0-65535.

Commento

28320: il numero o l'etichetta vengono posti in **H$**.

28325: se il primo carattere incontrato è una parentesi aperta '(' viene richiamata la routine per ottenere il numero concreto.

28335: all'ingresso nella routine alla linea 28150, la variabile T viene posta uguale a meno uno. Se resta inalterata, non è stato trovato alcun carattere significativo. Gli altri valori di T possono indicare uno spazio, due punti, punto e virgola, parentesi chiusa o punto. Se qualcuno di questi è associato ad una HS di lunghezza zero allora nell'istruzione non c'è un numero o un'etichetta ed il programma ritorna dalla subroutine.

28340: se non è presente alcuno dei delimitatori cercati nella linea precedente, si suppone che il numero sia preceduto da '$', '%' o '&', indicanti esadecimale, ottale o binario e l'esecuzione passa alla routine che estrae da essi un numero decimale.

28350: se il primo carattere è una cifra, viene preso il valore (VAL) di H$ – perciò le variabili non devono iniziare con un numero, dal momento che il valore del numero verrebbe prelevato e il resto del nome della variabile ignorato.

28360-28380: se il primo carattere non è una cifra allora si suppone che quanto è stato prelevato sia un'etichetta e si passa alla routine in linea 28250 per determinarne il valore.

28390-28490: il puntatore rappresentato da T ha ora passato un carattere che a questo punto si presume indichi una differente base numerica. Si verifica quest'ipotesi. Se viene specificata una base differente, la variabile BASE viene modificata per tenerne conto e viene richiamata la routine del monitor che converte numeri non decimali in decimali. Se il carattere indicato da T2 non è un segnalatore di scambio di base allora esso non è valido e viene segnalato l'errore 'label is not alphanumeric' (etichetta non alfanumerica). Se è stata specificata una base diversa ma la rappresentazione è scorretta (es. il numero binario 101012) verrà segnalato un errore di tipo 'incorrect number base' (base non corretta) al ritorno dalla routine nella linea 11950.

28492: il puntatore principale, PTR, che indica il carattere successivo al termine dell'operatore corrente, viene decrementato di uno in modo da poter rieseguire la procedura sul resto della linea.

```
28500 REM********************************
28501 REM EVALUATE TERM WITH * OR /
28502 REM********************************
28510 GOSUB 28300 : TERM = H
28520 IF PTR>LEN(IN$) THEN RETURN
28530 IF T=42 THEN GOSUB 28300 : TERM =
INT(TERM*H) : GOTO 28520
28550 IF T<>47 THEN RETURN
28560 GOSUB 28300
28570 IF H=0 THEN TERM = 0 : EN = 15 : G
OTO 28000
28580 TERM = INT(TERM/H)
28590 GOTO 28520
```

Uno dei problemi di valutazione delle espressioni è costituito dalla precedenza, cioè dal determinare quale parte dell'espressione A B/C +D/E F deve venir valutata per prima. Il valutatore di espressioni può trattare la precedenza fra '+' '—' ' ' e '/', ma non quella imposta dall'uso delle parentesi. Le parentesi renderebbero inoltre più difficile distinguere il tipo di operando. Questo particolare modulo tratta le due operazioni a precedenza più alta, moltiplicazione e divisione.

Commento

28510: viene prelevato un valore usando il modulo precedente e lo si memorizza nella variabile TERM.

28530: se il carattere puntato da T è un segno di moltiplicazione, viene prelevato il valore immediatamente seguente e moltiplicato per TERM.

28550-28580: se il carattere indicato da T è una '/', cioè un segno di divisione, si esegue un test per controllare che il divisore non sia zero; se lo è viene segnalata 'division by zero' (divisione per zero). TERM viene ora diviso per il valore appena ottenuto in H.

TAVOLA DI CONTROLLO

```
28500 123     28501 20      28502 123
28510 150     28520 150     28530 162
28550 67      28560 170     28570 152
28580 93      28590 170
```

```
28600 REM********************************
28601 REM EVALUATE EXPRESSION
28602 REM********************************
28605 ERR = FALSE
28610 GOSUB 28500 : RESULT = TERM
28620 IFT=-1OR T=32 OR T=58 OR T=59 OR T
=41 OR T=46 OR PTR>LEN(IN$) THEN RETURN
28630 IF T=43 THEN GOSUB 28500 : RESULT
= INT(RESULT+TERM) : GOTO 28620
28640 IF T=45 THEN GOSUB 28500 : RESULT
= INT(RESULT-TERM) : GOTO 28620
28650 RESULT = 0 : EN = 4 : GOTO 28000
```

Questo modulo valuta gli operatori, a precedenza inferiore, di addizione e sottrazione.

Commento

28610: questo modulo non chiama direttamente il modulo principale alla linea 28300, ma il modulo precedente. Questo assicura che vengano esaminati i valori prima di essere restituiti per determinare se debbano prima venir moltiplicati o divisi per qualcos'altro. Così se l'espressione da valutare fosse A*B+C, A*B verrebbe valutata prima di passare il risultato a questo modulo.

28620: se si rileva un delimitatore dopo l'operando, non c'è più nulla da valutare.

28630-28640: se T indica che un segno più o meno segue il valore finora ottenuto, viene eseguito il calcolo corrispondente.

28650: se il carattere indicato da T non è un più o un meno, viene segnalato un errore di 'invalid operator' (operatore non lecito) — i segni per o diviso sarebbero stati trattati dal modulo precedente.

TAVOLA DI CONTROLLO

```
28600 123     28601 54      28602 123
28605 70      28610 47      28620 5
28630 226     28640 229     28650 81
```

Finalmente è finito. Dipende ora dalle vostre energie lo sviluppare ulteriormente il programma Mastercode – che è uno dei programmi singoli più estesi mai pubblicati su un libro per un microcomputer. Nel resto del libro esamineremo una serie di routine in codice macchina che potrete immettere usando l'assembler. Se avete altri libri sulla programmazione del 6502, potreste trovarvi utili routine da immettere. Non è una cattiva idea provare con una o due piccole routine prima di iniziare ad estendere il BASIC del 64 col resto del libro, se non altro per familiarizzarsi col funzionamento del programma.

Si raccomanda comunque la solita attenzione: assicuratevi di aver salvato il programma prima di tentare qualunque cosa col linguaggio macchina. Tutti possono commettere un errore – e dolersene dopo aver perso il programma!

Parte II

CAPITOLO 5
L'ESTENSORE DEL BASIC

Per raggiungere il nostro scopo di realizzare routine in linguaggio macchina per estendere il linguaggio BASIC del 64, è in primo luogo necessario capire come lavora realmente il BASIC, come un normale comando BASIC viene prelevato ed elaborato senza per ora considerare i comandi che vogliamo aggiungere.

Consideriamo per primo un comando BASIC molto semplice, come una linea del tipo: 1 GOTO 10. Quando premete RETURN per immettere la linea, essa viene esaminata dall'interprete BASIC e viene rilevato il fatto che contiene una parola chiave BASIC. Questa parola chiave viene ridotta ad un singolo byte nel file di programma. Tutte le parole BASIC hanno questi byte, o 'token', i cui valori sono compresi fra 128 e 202 (più 255 per π). Nel caso del GOTO il token è 137.

Quando il programma contenente questa linea viene fatto eseguire, l'interprete BASIC esamina la linea, tralasciando il numero di linea fino a trovare il token, che viene riconosciuto come tale dal momento che è superiore a 128 e non racchiuso fra virgolette. Il token indica una posizione in una tabella ed in quella posizione si trova l'indirizzo di una routine in linguaggio macchina che eseguirà il comando espresso con GOTO 10. L'interprete esegue ora questa routine macchina che per prima cosa esamina la linea seguente il token di GOTO per ottenere un numero di linea. Avutolo dalla linea BASIC, il resto della routine macchina per questo particolare comando si occupa di trovare la linea cui ci si riferisce e di alterare un certo numero di variabili di sistema in modo che l'esecuzione salti a quel punto. Se non si trova alcun numero dopo il GOTO viene indicato un errore sintattico. Se viene trovato un numero di linea ma essa non fa parte del programma viene segnalato un errore di 'linea non definita'. Supponendo che tutto sia andato come previsto, il controllo del programma ora torna alla parte di interprete BASIC che ha il compito di cercare il prossimo token nel programma.

Da tutto questo possiamo ricavare un certo numero di azioni necessarie per l'esecuzione di una parola chiave BASIC:

1) l'interprete deve riconoscerla come parola chiave ed essere in grado di ridurla alla forma di token;

2) l'interprete dev'essere in grado di riconoscere il token una volta che il programma vada in esecuzione;

3) dev'esserci in memoria una tabella da cui l'interprete possa ricavare l'indirizzo iniziale della routine macchina che esegue il comando;

4) la routine macchina può dover essere in grado di ricavare ulteriori informazioni per il comando (es. il '10' per GOTO 10);

5) dev'esserci la possibilità di riconoscere e segnalare errori evitando l'esecuzione della routine.

Dopo aver fatto eseguire il programma, c'è un altro requisito imposto dal LIST. Non serve cercare di stampare i token invece delle parole chiave. L'interprete deve avere anche una tabella che gli permetta di ricavare la parola corrispondente a ciascun token, in modo da stamparla nel listing del programma.

Per poter introdurre nuove parole dobbiamo tener conto di tutto ciò. Una parola chiave deve venir posta nell'interprete, dev'essere specificato un token per essa, devono essere fornite le routine corrispondenti e, cosa più importante, l'interprete dev'essere forzato a riconoscere ed elaborare l'informazione data. Tutto ciò comporterà chiaramente la modifica dell'interprete BASIC, cosa che rappresenta da sè il primo problema, dal momento che, come senza dubbio sapete, l'interprete non è in una zona di memoria modificabile (RAM), ma nei chip il cui contenuto è fissato al momento della fabbricazione. Tutto ciò è vero, ma fortunatamente non è tutta la verità.

Quando accendete il 64, i suoi 64K di memoria sono occupati (grosso modo) da 8K di memoria per il Kernal (un insieme di routine macchina molto utili e comuni a quasi tutte le macchine Commodore), 8K per l'interprete BASIC, 1K per le variabili di sistema (locazioni che il 64 usa per memorizzare valori ed indirizzi importanti per le sue operazioni), 4K di RAM che non può essere usata dal BASIC e 4K per gestire altri dispositivi come il chip VIC, dischi, nastri, stampanti ecc. Comunque è possibile passare alcune sezioni di questa memoria al controllo dell'utente (RAM) – di fatto sono disponibili 64K di memoria volendo escludere ogni altra cosa, cioè il BASIC, le comunicazioni con l'esterno ecc.

L'interprete BASIC sembra occupare la memoria dalla posizione 40960 in avanti. Di fatto il 64 fa credere alla CPU che il chip separato dall'interprete BASIC occupi quella posizione. I veri 8K di RAM in questa zona sono totalmente inutilizzati, per il semplice motivo che il chip 6502/6510 può vedere solo 64K di memoria alla volta, così se deve vedere l'interprete BASIC non vedrà gli 8K di memoria utente corrispondenti. L'importanza di questo per noi è che ci sono 8K inutilizzati, esatta-

mente lo spazio per contenere l'interprete BASIC se dovesse risiedere su RAM invece che su ROM ed esattamente nel punto in cui la CPU si aspetterebbe di trovare l'interprete BASIC.

Il nostro primo passo nell'alterare l'interprete BASIC è perciò di ricopiare il contenuto dell'interprete BASIC in quell'area. Questo ha i suoi svantaggi — ora l'interprete può venir alterato (erroneamente) in modo casuale, cosa che non potrebbe accadere se fosse tenuto in ROM. Ma può anche essere modificato positivamente se sappiamo quello che facciamo!

Qui sotto è presentato un breve programma che pone tre routine macchina in memoria. L'effetto di queste è quello di spostare l'interprete BASIC in RAM. Più tardi verranno eseguite brevi aggiunte per permetterci di aggiungere nuove parole chiave e fare gli altri necessari cambiamenti. Per il momento basterà spostare l'interprete.

BASIC Extender - Listing I

```
0 REM12345678901234567890123456789012345
67890123456789012345678901 23
100 REM BASIC EXTENDER ROUTINE
110 REM MOVE BASIC ROM INTO RAM AT $A000
 - $BFFF
120 DATA 165,1,41,254,133,1,96,160,255,2
00,32,32,8,190,1,1,32,6,8,138,153,1,1
130 DATA 200,208,240,165,1,9,1,133,1,96,
32,6,8,24,162,255,232,160,255,200,185
135 DATA 75,8,133,20,185,151,8,48,01,96,
133,21,185,227,8,129,20,144,235
139 DATA 0
140 AD = 2054
150 READ A : IF A<>0 THEN POKE AD,A : AD
 = AD+1 : GOTO 150
160 REM DO ACTUAL MOVE
165 POKE 2068,0 : POKE 2075,0
170 FOR X = 160 TO 191
190 POKE 2069,X : POKE 2076,X
200 SYS 2061
210 NEXT
220 POKE 2068,1 : POKE 2075,1
230 SYS 2054
300 END
```

Commento

0-150: lo scopo delle frasi REM è quello di fornire un'area di memoria pulita. In quest'area, che inizia alla locazione 2054 (se fate una PEEK di 2053 vi troverete il token del primo REM, cioè 143), la linea 150 immette con l'istruzione POKE i valori contenuti nelle frasi DATA. Le frasi DATA rappresentano, come avrete senz'altro capito, istruzioni macchina.

Routine in linguaggio macchina

```
add.   data      source code
0                10 PRT
0                20 SYM
0                30 ORG 2054
806    A501      50 LBL000 LDA 1
808    29FE      60 AND #254
80A    8501      70 STA 1
80C    60        80 RTS
80D    A0FF      90 LDY #255
80F    C8        100 INY
810    202008    120 LBL001 JSR LBL002
813    BE0101    130 LDX 257.Y
816    200608    140 JSR LBL000
819    8A        150 TXA
81A    990101    160 STA 257.Y
81D    C8        170 INY
81E    D0F0      180 BNE LBL001
820    A501      190 LBL002 LDA 1
822    0901      200 ORA #1
824    8501      210 STA 1
826    60        220 RTS
```

Commento sulle routine macchina

Qui sopra è fornito il listing delle routine usate nella forma del Mastercode assembler, per chiarezza. NON IMMETTETE LE ROUTINE USANDO L'ASSEMBLER O LO ROVINERETE. USATE IL PROGRAMMA DATO SOPRA.

10-30: direttive assembler che mandano l'uscita alla stampante, stampano la tavola simboli e fanno iniziare il programma alla locazione 2054 in memoria.

50-80: queste istruzioni caricano in accumulatore il contenuto del byte 1 della memoria, ne fanno l'AND con 254 (ponendo così a zero il bit zero) e infine memorizzano il risultato nella locazione uno. Questo provoca lo 'spegnimento' della ROM e l''accensione' della RAM corrispondente. La routine è auto-consistente. Si sarebbe potuto ottenere lo stesso effetto con l'istruzione BASIC POKE 1, PEEK (1) AND 254, ma questo avrebbe interrotto il funzionamento del sistema dal momento che non ci sarebbe più stato un interprete BASIC su cui lavorare.

90-180: questa routine inizia con un salto alla terza sezione del programma che 'accende' la ROM e 'spegne' la RAM. A questo punto del programma la linea 130 non ha senso; più avanti l'indirizzo da cui verrà caricato il registro X verrà immesso tramite una POKE nel codice macchina. L'effetto finale della routine è di caricare nel registro X un byte dell'interprete, 'spegnere' la ROM, trasferire il contenuto di X all'accumulatore e memorizzare questo byte dalla ROM nello stesso indirizzo, ma con la RAM ora 'accesa'. Una volta fatto ciò viene incrementato Y. Quando Y raggiunge 255 (cioè sono stati trasferiti 256 byte) l'istruzione BNE non verrà più eseguita e si passerà alla routine seguente.

190-220: immagine speculare di 50-80, queste linee 'riaccendono' la ROM prima di restituire il controllo al BASIC.

Noterete che questo listing non comprende tutto ciò che avete immesso — il resto verrà spiegato più avanti, al momento non viene usato.

Tornando al programma BASIC in sè:

165: i nostri programmi in linguaggio macchina richiederanno l'uso di due zeri, visto che gli indirizzi su cui si baseranno le nostre mosse di 256 byte avranno tutti uno zero nel byte basso es. 40960 che è 160, e zero nella rappresentazione a due byte. Mettere gli zeri permanentemente nelle frasi REM avrebbe creato grossi problemi per l'immissione di nuove linee. Incontrando gli zeri all'immissione di nuove linee la routine BASIC 'collegamento di linee' avrebbe rovinato le frasi REM, cercando di romperle dove fossero presenti gli zeri. Per questo motivo, gli zeri vengono temporaneamente immessi con POKE per gli scopi del linguaggio macchina e poi rimpiazzati da uni alla linea 220.

165-200: i POKE danno alle istruzioni macchina alle linee 130 e 160 del listing in assemblatore gli indirizzi di partenza di 32 blocchi da 256 byte (8K in totale) che verranno passati dalla ROM alla RAM. Quando l'indirizzo di ciascun blocco è stato immesso via POKE nel programma in linguaggio macchina, il programma viene fatto eseguire dalla chiamata SYS alla linea 200, che fa partire la routine macchina

situata alla linea 120, del listing in linguaggio assemblatore. Vengono spostati 256 byte quindi viene immesso l'indirizzo del blocco successivo.

Se avete battuto il programma e l'avete controllato, salvatelo prima che sia troppo tardi. Ora battete SYS 2054 in modo diretto. Se avete commesso un errore quasi certamente la macchina si incepperà. Altrimenti sembrerà che non sia successo nulla.

Ora provate questo:

300 END (RETURN).......ancora nulla

POKE 41118,72 (RETURN)........ancora nulla

LIST (RETURN)....date un'occhiata all'ultima riga. Dovrebbe leggersi 300 HND – altrimenti avete fatto un errore e dovete spegnere, riaccendere a far ripartire il programma BASIC che avete salvato. Se la procedura ha funzionato, immettendo END (RETURN) provocherete un messaggio 'SYNTAX ERROR', cosa che non accadrà con un comando HND. Ciò che avete fatto è stato alterare la tabella contenente le parole chiave BASIC. Se vole proprio divertirvi, battete RUN/RESTORE per 'riaccendere' la ROM, caricate il monitor ed esaminate la memoria a partire da 41118 (A09E esadecimale) in avanti. Vi troverete le locazioni delle parole chiave, ciascuna apparentemente priva dell'ultimo carattere. Prendete nota (o fate stampare) delle locazioni poi ricaricate questo programma e rilocate l'interprete. Se non modificate il carattere finale o la lunghezza di una parola chiave, potete far leggere tutto ciò che volete. Provate e poi listate il programma o fatelo stampare. Funzionerà ancora, ma il suo strano aspetto proverà che avete iniziato a dimostrare il vostro potere sul BASIC.

CAPITOLO 6
BASIC E CODICE MACCHINA

Listing in linguaggio assembler

```
 10         PRT
 20         ORG $C000
 30         SYM
 40         ;----------------------------
--------
 50         KEYWRD
 60         ; NEW FUNCTION KEYWORDS
 70         ; DEEK
 80         BYT 68.69.69.75+128
 90         ; YPOS
 100         BYT 89.80.79.83+128
 110         ; VARPTR
 120         BYT 86.65.82.80.84.82+128
 130         ;---------------------------
--------
 140         ; NEW ACTION KEYWORDS
 150         ; DOKE
 160         BYT 68.79.75.69+128
 170         ; RKILL
 180         BYT 82.75.73.76.76+128
 190         ; DELETE
 200         BYT 68.69.76.69.84.69+128
 210         ; MOVE
 220         BYT 77.79.86.69+128
 230         ; FAST
 240         BYT 70.65.83.84+128
 250         ; SLOW
 260         BYT 83.76.79.87+128
 270         ; PLOT
```

```
 280         BYT 80.76.79.84+128
 290         ; UNDEAD
 300         BYT 85.78.68.69.65.68+128
 310         ; SUBEX
 320         BYT 83.85.66.69.88+128
 330         ; BLOAD
 340         BYT 66.76.79.65.68+128
 350         ; BVERIFY
 360         BYT 66.86.69.82.73.70.89+128
 370         ; BSAVE
 380         BYT 66.83.65.86.69+128
 390         ; FILL
 400         BYT 70.73.76.76+128
 410         WRD 0
 420         ; THIS IS THE END OF KEYWORD
TABLE CHR.
 430         ;-----------------------------
--------
 440         ; NEW ACTION VECTOR
 450         ACTVEC
 460         WRD $A830
 470         WRD $A830
 480         WRD $A830
 490         WRD $A830
 500         WRD $A830
 510         WRD $A830
 520         WRD $A830
 530         WRD $A830
 540         WRD $A830
 550         WRD $A830
 560         WRD $A830
 570         WRD $A830
 580         WRD $A830
 590         ;-----------------------------
--------
 600         ; NORMAL IS THE NORMAL NUMBER
 OF BASIC KEYWORDS
 610         NORMAL = 75
 620         ; NEWACT IS THE NUMBER OF NEW
 ACTION KEYWORDS
 630         NEWACT = 13
 640         ; NEWFUN IS THE NUMBER OF NEW
 FUNCTION KEYWORDS
 650         NEWFUN = 3
 660         ; USE BY POKEING A7E1 WITH 'J
MP EXECUTE'
```

```
 670        EXECUT JSR $73
 680        JSR DOEX
 690        JMP $A7AE
 700        DOEX BEQ LABEL
 710        SBC #$80
 720        BCC DOLET
 730        CMP #NORMAL+NEWFUN+1
 740        BCC RETURN
 750        CMP #NORMAL+NEWFUN+NEWACT+1
 760        BCS RETURN
 770        ; EXECUTE THE NEW ACTION KEYW
ORDS
 780        SBC #NORMAL+NEWFUN
 790        ASL A
 800        TAY
 810        LDA ACTVEC+1.Y
 820        PHA
 830        LDA ACTVEC.Y
 840        PHA
 850        JMP $73
 860        LABEL RTS
 870        DOLET JMP $A9A5
 880        RETURN JMP $A7F3
 890        ;------------------------------
--------
 900        ; PRINT TOKEN ROUTINE TO USE
POKE 774 & 775 WITH PRTTOK ADDRESS
 910        PRTTOK JSR PUTREG
 920        CMP #NORMAL+129
 930        BCC PRTNOR
 940        ; PRINT THE NEW TOKENS
 950        LDA ASAVE
 960        SBC #NORMAL+1
 970        STA ASAVE
 980        LDA #KEYWRD/256
 990        LDX #KEYWRD-KEYWRD/256*256
 1000        JMP LBL000
 1010        ; PRINT NORMAL TOKENS
 1020        PRTNOR LDA #$A0
 1030        LDX #$9E
 1040        LBL000 STA $A732
 1050        STX $A731
 1060        STA $A73A
 1070        STX $A739
 1080        JSR GETREG
 1090        JMP $A71A
 1100        ;------------------------------
```

```
----------
 1110        ; CRUNCH TOKENS ROUTINE EXTR
A CODE
 1120        ; USE BY ALTERING $A604 TO '
JMP CRUNCH'
 1130        CRUNCH JSR PUTREG
 1140        LDA $A5FC
 1150        CMP #$A0
 1160        BNE STAND
 1170        LDA #$C0
 1180        LDX #$00
 1190        JSR TOKSTR
 1200        JSR GETREG
 1210        LDY #0
 1220        JMP $A5B8
 1230        STAND LDA #$A0
 1240        LDX #$9E
 1250        JSR TOKSTR
 1260        JSR GETREG
 1270        LDA $200.X
 1280        JMP $A607
 1290        ;---------------------------
----------
 1300        GETREG LDA PSAVE
 1310        PHA
 1320        LDA ASAVE
 1330        LDX XSAVE
 1340        LDY YSAVE
 1350        PLP
 1360        RTS
 1370        ;---------------------------
----------
 1380        PUTREG PHP
 1390        STA ASAVE
 1400        STX XSAVE
 1410        STY YSAVE
 1420        PLA
 1430        STA PSAVE
 1440        LDA ASAVE
 1450        RTS
 1460        ;---------------------------
----------
 1470        TOKSTR STA $A5BE
 1480        CLD
 1490        STX $A5BD
```

```
 1500         STA $A601
 1510         STX $A600
 1520         DEX
 1530         CPX #$FF
 1540         BNE LBL003
 1550         SEC
 1560         SBC #1
 1570         LBL003 STA $A5FC
 1580         STX $A5FB
 1590         RTS
 1600         ;-----------------------------
----------
 1610         PSAVE
 1620         ASAVE = PSAVE+1
 1630         XSAVE = ASAVE+1
 1640         YSAVE = XSAVE+1

END
```

Nota: questo listing è riportato in forma completamente assemblata al termine del commento ma è presentato qui in forma non assemblata per amore di chiarezza.

Ora sappiamo che siamo in grado di alterare il BASIC. Comunque rendere strane le parole chiave non è il nostro scopo. Vogliamo invece estendere il BASIC e per far ciò dobbiamo eseguire delle modifiche un po' più consistenti che qualche POKE qua e là. Passiamo perciò ad esaminare il più esteso programma macchina che dovrete trascrivere nel corso di questo libro; sfortunatamente dev'essere fatto a questo punto o nulla del resto funzionerebbe.

Ricorderete che nello scorso capitolo abbiamo detto che per elaborare un comando l'interprete BASIC dev'essere in grado di riconoscere il token generato durante il listing del programma ed anche di sapere dov'è situata la routine macchina che esegue l'azione che la parola chiave specifica. Questo viene fatto con l'aiuto di

una tabella situata in memoria alle locazioni da A09E a A19D (41118-41373). Quando l'interprete BASIC incontra una parola chiave in ingresso, esamina la tabella fino a trovare una corrispondenza, altrimenti viene segnalato un errore sintattico. Se la parola chiave si trova nella tabella, la sua posizione (in tabella, non in memoria – ad es. la parola chiave n. 0 è END) sommata a 128 dà il token corretto per essa, così il token per END è 128. Quando un programma viene eseguito il valore del token (–128) viene moltiplicato per due e quindi interpretato come posizione in un'altra tabella; la tabella dei vettori. Questa tabella dice all'interprete BASIC dove trovare la routine macchina per quella particolare istruzione.

I problemi che chiunque voglia estendere il BASIC si trova ad affrontare sono perciò:

1) ricerca di spazio per le nuove parole chiave nella tabella delle parole chiave;

2) estensione della tabella dei vettori per indicare gli indirizzi delle nuove routine macchina;

3) ultimo, ma non meno importante: immissione delle nuove routine.

Questi problemi sarebbero quasi banali se ci fosse spazio nelle tabelle delle parole chiave e dei vettori. In effetti, la tabella delle parole chiave viene esaminata sotto il controllo di un singolo registro della CPU (il registro Y). Il registro è costituito da un solo byte, quindi può trattare solo un'area di 256 byte di lunghezza. Le parole chiave esistenti nel BASIC del 64 riempiono esattamente questi 256 byte (compreso un misterioso comando 'GO' non menzionato nel manuale che permette di separare GO e TO trattandoli ugualmente come GOTO). In altre parole non si possono fare aggiunte alla tabella. Inoltre se anche si potesse non c'è spazio nella tabella dei vettori per gli indirizzi di nuove routine. In memoria c'è spazio a volontà per le nuove routine, ma come fare a far riconoscere all'interprete le parole chiave corrispondenti?

Qual è la risposta? Come in altre occasioni, quando avete a disposizione un programma che non volete modificare troppo, in questo caso l'interprete BASIC, la risposta è che è necessario barare. Quando riduce una parola chiave ad un token, l'interprete esamina la tabella delle parole chiave finché non trova uno zero (l'ultimo byte della tabella). Se si raggiunge questo punto, la parola chiave non è stata trovata e verrà indicato un errore di sintassi. Il nostro metodo per estendere la tabella delle parole chiave è sostituire l'istruzione macchina seguente alla rilevazione dello zero con un salto ad una nuova subroutine che inizia nuovamente la ricerca ad un nuovo indirizzo, fornendo così una tabella vuota di 256 byte in più in cui me-

morizzare le parole chiave. Questo trucchetto risolve il problema della 'riduzione' delle parole chiave, resta solo da convincere l'interprete a riconoscere i tokens generati alla scoperta di una parola chiave nella nuova tabella.

L'esecuzione dei singoli comandi BASIC è controllata da una sezione dell'interprete il cui scopo è di consultare la tabella dei vettori e, sulla base delle informazioni contenutevi, di richiamare la subroutine appropriata per il particolare token. La routine RUN, che controlla l'intera esecuzione del programma, richiama questa routine di 'comando singolo' ogni volta che viene trovato un token nel programma. Fortunatamente per noi, la routine RUN non ha l'indirizzo della routine 'comando singolo' incorporato, ma lo deve leggere da una locazione di memoria RAM (308-309 esadecimale).

Per assicurarci che i nuovi token vengano letti ed elaborati alterneremo l'indirizzo indicato dagli esadecimali 308-309, in modo che indichi una nuova routine di esecuzione di comando singolo progettata da noi. La nuova routine prima controlla se il token incontrato è quello di una delle nostre nuove parole chiave (che possono essere riconosciute dal fatto che il loro valore è maggiore o uguale a CC esadecimale-204 decimale). Se non si tratta di uno dei nuovi comandi, l'esecuzione passa alla routine normale di comando singolo. Se invece è uno dei nuovi comandi, allora la nostra routine prende il controllo, indicando un indirizzo completamente nuovo per la tabella dei vettori. In questo modo possiamo aggiungere una nuova tabella dei vettori all'interprete oltre che una nuova tabella delle parole chiave.

Se siete riusciti a seguire abbastanza questa spiegazione, siete pronti per proseguire e dare un'occhiata alla routine macchina centrale, essenziale per i cambiamenti da fare.

Modulo 10-30: queste tre linee rappresentano istruzioni per l'assembler che dovreste riconoscere dal commento alla sezione assembler del programma Mastercode. PRT indica che il listing dovrà essere mandato alla stampante (dal comando PRT in poi), ORG dice all'assembler di iniziare a memorizzare il programma a partire dalla linea C000 e SYM assicura che venga stampata la tavola simboli alla fine del listing. Questa routine macchina e le seguenti occuperanno l'area libera di RAM fra C000 e CFFF(49152-53247). Quest'area di memoria non è a disposizione del BASIC ed è così una posizione ideale per memorizzare codice macchina.

Modulo 40-120: questa sezione del programma estende la tabella delle parole chiave con tre nuove parole, DEEK, YPOS e VARPTR – spiegheremo poi di cosa si tratta. A questo punto basta che sia chiara la differenza fra funzioni ed azioni in BASIC. Una funzione è un'operazione matematica che deve ricorrere dopo un '='

o un IF, ad esempio, a differenza di un'azione, che può trovarsi isolata e non può far parte di un'espressione matematica. La ragione per cui è importante distinguere le due cose è che azioni e funzioni vengono trattate diversamente durante l'esecuzione di un programma. Ad esempio viene chiamata una routine speciale ogni volta che si trova un segno di uguale. A causa di questo trattamento speciale, l'interprete tiene conto della posizione delle parole chiave per funzioni nella tabella delle parole chiave ricordando dove iniziano e dove terminano. Le parole al di fuori di questo intervallo vengono trattate come azioni e viene segnalato un errore di sintassi se si incontrano all'interno di un'espressione, come ad esempio LET A = POKE 123,123. Nella tabella normale delle parole chiave, quelle corrispondenti a funzioni si trovano tutte alla fine, così la posizione più semplice in cui situare nuove parole chiave funzioni è l'inizio della nuova tabella, visto che ciò comporterà solo la modifica del puntatore alla fine delle parole chiave per le funzioni (in questo caso dovrà essere incrementato di tre).

In questo modulo le sole linee operative, cioé quelle che verranno assemblate in memoria, sono quelle con le istruzioni BYT. Esse definiscono i caratteri delle nuove parole chiave in ASCII con l'aggiunta di 128 al valore dell'ultimo carattere per segnalare il termine della parola (il bit 7 dell'ultimo carattere sarà perciò uguale ad 1).

Modulo 130-420: in questo modulo sono introdotte le nuove parole chiave per le azioni con cui estenderemo il BASIC. Il formato della tabella è esattamente lo stesso spiegato nel modulo precedente con l'eccezione che la lista delle parole chiave termina con un'istruzione che assicura che la tabella termini con uno zero. Questo permetterà all'interprete di rilevare il termine della tabella.

Modulo 430-580: questa è la nuova tabella dei vettori. A questo momento non avete ancora introdotto alcuna delle routine che permettono alle nuove parole chiave di fare alcunché. Immettendo questo programma assicurate soltanto che le parole nuove possano essere usate nei programmi BASIC, trasformate in token e venir rilistate. Se vengono incontrate in esecuzione dopo che sia stata immessa questa sezione di programma macchina, tutto ciò che succederà è che il programma terminerà, dal momento che i loro vettori puntano inizialmente alla routine END del BASIC normale. Notate che non vengono inclusi vettori per le parole corrispondenti a funzioni: esse verranno trattate da una routine separata di valutazione delle espressioni che esporremo più avanti.

Modulo 590-650: questo modulo predispone tre etichette con il numero delle parole chiave BASIC normali, il numero delle nuove parole chiave per azioni ed il numero delle nuove parole chiave per funzioni. Queste verranno usate per rendere più

leggibile il resto di questa sezione di codice e rendere più facili eventuali modifiche se vorrete aggiungere vostre nuove parole chiave.

Modulo 660-880: questo modulo si occupa dell'esecuzione delle nuove parole chiave BASIC.

Commento

660-690: ricorderete che abbiamo già spiegato che, al fine di assicurarci che i nostri token non vengano rifiutati dall'interprete, modifichiamo l'indirizzo (in 308-309 esadecimali) che la routine RUN usa per chiamare la routine di esecuzione di un comando singolo (SCER). Di fatto le locazioni 308-309 rinviano l'esecuzione del programma ad un comando di salto nella routine RUN ed è questo salto che chiama la routine SCER. Il riferimento agli esadecimali 308-309, che si trovano naturalmente in RAM, è un'intelligente previsione della Commodore per permettere ai pogrammatori di fare proprio quello che stiamo facendo noi, cioè di sostituire la SCER standard. Queste tre linee sostituiscono tre linee sottratte alla routine RUN, inclusa una chiamata alla routine che preleva il prossimo carattere di una linea, un salto alla (nostra) SCER ed un salto indietro all'inizio della routine RUN.

700-760: queste linee determinano se si è raggiunto il termine di una linea (cioè se è stato rilevato uno zero nel salto a $73), controllano il valore di una variabile (minore di $80) ed infine esaminano il valore di quello che, a questo punto, deve essere un token per vedere se rientra nell'intervallo della nuova tabella di parole chiave delle azioni. In caso negativo, l'esecuzione passa al normale interprete.

770-850: il valore della fine della nuova tabella delle funzioni (cioè il numero delle parole chiave originali più il numero delle nuove funzioni) viene sottratto dal valore del token, ottenendone così la posizione nella nuova tabella delle parole chiave per azioni. Viene usato il registro Y per ricavare il corrispondente vettore di azione nella tabella dei vettori. L'indirizzo viene inserito in cima alla pila. Ora facciamo un salto alla routine alla posizione $73, che preleva il carattere seguente nella linea (questo perché tutte le routine dell'interprete lavorano con la convenzione che il carattere sia stato posto in accumulatore anche se non ne hanno bisogno effettivamente per le loro operazioni). Alla fine della subroutine chiamata in $73 c'è un'istruzione RTS che preleva il vettore azione dalla pila interpretandolo come indirizzo di ritorno. Notate che il ritorno effettivo va fatto all'indirizzo successivo a quello presente in pila per evitare di tornare alla istruzione che ha appena chiamato la subroutine. Ciò significa che tutti i vettori azione della tabella puntano al byte precedente la routine che vogliono richiamare.

860-880: queste linee etichettate fanno varie chiamate se i test durante il programma sono 'falliti' – vengono richiamate dalle quattro istruzioni di BRANCH precedenti.

Modulo 890-1090: questo ripristina le parole a partire dai token quando il programma viene listato e sarà chiamato dalla normale routine di LIST dopo alcune modifiche.

Commento

910-930: dal momento che questa sezione viene richiamata nel mezzo dell'esecuzione di un'altra routine (LIST), vengono prima salvati i registri in modo che possano essere ripristinati al ritorno. Il token prelevato nell'accumulatore da LIST viene confrontato col massimo valore ammissibile per un normale token. Se ha valore minore, viene eseguita la normale routine di ricostruzione della parola chiave.

950-1000: il valore del token in accumulatore viene modificato sottraendo il numero dei normali token. L'accumulatore indicherà ora una posizione nella nuova tabella e questo valore viene memorizzato in ASAVE. I registri A ed X vengono caricati con l'indirizzo iniziale della nuova tabella delle parole chiave e quest'indirizzo viene poi posto nella normale routine di traduzione dei token da una chiamata alla LBL000.

1010-1090: se dev'essere stampata una parola chiave normale, l'indirizzo della tabella normale delle parole chiave viene inserito nella routine di traduzione dei tokens, che può aver lavorato in precedenza sulla nuova tabella.

Modulo 1100-1280: abbiamo già detto che dopo aver rilevato la fine della tabella normale delle parole chiave, la routine di creazione dei token viene fatta passare ad una nuova nostra routine, che è la presente. Lo scopo del modulo è di creare i nuovi token.

Commento

1140-1160:$A5FC contiene il byte alto dell'indirizzo della tabella corrente delle parole chiave, che può essere quella normale o la nostra. Se il valore è $A0 allora è stata letta la fine della tabella normale e viene eseguita la sezione successiva.

1170-1220: l'indirizzo della nuova tabella delle parole chiave viene posto nella routine di creazione dei token (JSR TOKSTR) e la routine viene rieseguita, caricando 0 nel registro Y per assicurare che cominci all'inizio della nuova tabella. Al termine della seconda ricerca, CRUNCH (linea 1130) verrà richiamata ma questa volta sarà eseguita l'istruzione BNE STAND.

1230-1280: l'indirizzo della tabella normale delle parole chiave viene reinserito nella routine di creazione dei token, i registri vengono ripristinati (JSR GETREG) e quindi l'istruzione che era stata cancellata per chiamare questa routine viene riscritta. Infine l'esecuzione passa alla routine di creazione dei token che completa l'opera.

Modulo 1290-1360: questo modulo ripristina i valori dei registri che sono stati salvati dal modulo seguente. L'unica complicazione è data dal ripristino del registro di stato del processore, che non può essere caricato direttamente dalla memoria. L'operazione è realizzata portando il valore in accumulatore, mettendolo in pila e infine reimmettendolo nel registro del processore.

Modulo 1370-1450: l'opposto del modulo precedente, vengono salvati i contenuti dei registri. Notate ancora l'uso della pila e dell'accumulatore per salvare il contenuto del registro di stato.

Modulo 1460-1590: l'indirizzo della tabella delle parole chiave è tenuto in tre distinte locazioni nella routine di creazione dei token. Questo modulo piazza l'indirizzo richiesto (tabella nuova o normale) in quelle locazioni con la leggera complicazione che la locazione finale richiede il byte precedente la tabella, così si sottrae 1 dall'indirizzo nelle linee 1520-1560 prima di memorizzarlo.

Modulo 1600-1640: queste linee non interesseranno la memoria quando il programma sarà assemblato. Esse dicono all'accumulatore di inizializzare le variabili specificate dal programma per l'uso.

Sommario

A questo punto vi dispiacerà sentire che se avete immesso il programma nell'assembler e l'avete assemblato tutto ciò che potete fare è salvarlo per qualche tempo. Prima che i cambiamenti introdotti da questo codice possano avere buon esito, si

devono fare un paio di modifiche al BASIC extender presentato sopra. Qui in basso c'è il listing del codice come dovrebbe risultare una volta assemblato. Controllate il vostro programma con questo, paragonando valori dei BYTES e indirizzi: spesso i programmi in linguaggio macchina che non funzionano sono fonte di ispirazione.

CODICE MACCHINA: Listato completamente assemblato

```
add.    data      source code
0                 10 PRT
0                 20 ORG $C000
C000              30 SYM
C000              40 ;-------------------
----------------
C000              50 KEYWRD
C000              60 ; NEW FUNCTION KEYWORD
S
C000              70 ; DEEK
C000    444545    80 BYT 68.69.69.75+128
C004              90 ; YPOS
C004    59504F    100 BYT 89.80.79.83+128
C008              110 ; VARPTR
C008    564152    120 BYT 86.65.82.80.84.82
+128
C00E              130 ;-------------------
----------------
C00E              140 ; NEW ACTION KEYWORDS
C00E              150 ; DOKE
C00E    444F4B    160 BYT 68.79.75.69+128
C012              170 ; RKILL
C012    524B49    180 BYT 82.75.73.76.76+12
8
C017              190 ; DELETE
C017    44454C    200 BYT 68.69.76.69.84.69
+128
C01D              210 ; MOVE
C01D    4D4F56    220 BYT 77.79.86.69+128
C021              230 ; FAST
```

```
C021  464153    240 BYT 70.65.83.84+128
C025            250 ; SLOW
C025  534C4F    260 BYT 83.76.79.87+128
C029            270 ; PLOT
C029  504C4F    280 BYT 80.76.79.84+128
C02D            290 ; UNDEAD
C02D  554E44    300 BYT 85.78.68.69.65.68
+128
C033            310 ; SUBEX
C033  535542    320 BYT 83.85.66.69.88+12
8
C038            330 ; BLOAD
C038  424C4F    340 BYT 66.76.79.65.68+12
8
C03D            350 ; BVERIFY
C03D  425645    360 BYT 66.86.69.82.73.70
.89+128
C044            370 ; BSAVE
C044  425341    380 BYT 66.83.65.86.69+12
8
C049            390 ; FILL
C049  46494C    400 BYT 70.73.76.76+128
C04D  0000      410 WRD 0
C04F            420 ; THIS IS THE END OF
KEYWORD TABLE CHR.
C04F            430 ;--------------------
----------------
C04F            440 ; NEW ACTION VECTOR
C04F            450 ACTVEC
C04F  30A8      460 WRD $A830
C051  30A8      470 WRD $A830
C053  30A8      480 WRD $A830
C055  30A8      490 WRD $A830
C057  30A8      500 WRD $A830
C059  30A8      510 WRD $A830
C05B  30A8      520 WRD $A830
C05D  30A8      530 WRD $A830
C05F  30A8      540 WRD $A830
C061  30A8      550 WRD $A830
C063  30A8      560 WRD $A830
C065  30A8      570 WRD $A830
C067  30A8      580 WRD $A830
C069            590 ;--------------------
----------------
C069            600 ; NORMAL IS THE NORMA
L NUMBER OF BASIC KEYWORDS
C069            610 NORMAL = 75
```

```
C069            620 ; NEWACT IS THE NUMBE
R OF NEW ACTION KEYWORDS
C069            630 NEWACT = 13
C069            640 ; NEWFUN IS THE NUMBE
R OF NEW FUNCTION KEYWORDS
C069            650 NEWFUN = 3
C069            660 ; USE BY POKEING A7E1
 WITH 'JMP EXECUTE'
C069  207300    670 EXECUT JSR $73
C06C  2072C0    680 JSR DOEX
C06F  4CAEA7    690 JMP $A7AE
C072  F01B      700 DOEX BEQ LABEL
C074  E980      710 SBC #$80
C076  9018      720 BCC DOLET
C078  C94F      730 CMP #NORMAL+NEWFUN+1
C07A  9017      740 BCC RETURN
C07C  C95C      750 CMP #NORMAL+NEWFUN+NE
WACT+1
C07E  B013      760 BCS RETURN
C080            770 ; EXECUTE THE NEW ACT
ION KEYWORDS
C080  E94E      780 SBC #NORMAL+NEWFUN
C082  0A        790 ASL A
C083  A8        800 TAY
C084  B950C0    810 LDA ACTVEC+1.Y
C087  48        820 PHA
C088  B94FC0    830 LDA ACTVEC.Y
C08B  48        840 PHA
C08C  4C7300    850 JMP $73
C08F  60        860 LABEL RTS
C090  4CA5A9    870 DOLET JMP $A9A5
C093  4CF3A7    880 RETURN JMP $A7F3
C096            890 ;----------------------
----------------
C096            900 ; PRINT TOKEN ROUTINE
 TO USE POKE 774 & 775 WITH PRTTOK ADDRE
C096  20FAC0    910 PRTTOK JSR PUTREG
C099  C9CC      920 CMP #NORMAL+129
C09B  900F      930 BCC PRTNOR
C09D            940 ; PRINT THE NEW TOKEN
S
C09D  AD29C1    950 LDA ASAVE
C0A0  E94C      960 SBC #NORMAL+1
C0A2  8D29C1    970 STA ASAVE
C0A5  A9C0      980 LDA #KEYWRD/256
C0A7  A200      990 LDX #KEYWRD-KEYWRD/25
6*256
```

```
C0A9  4CB0C0    1000 JMP LBL000
C0AC            1010 ; PRINT NORMAL TOKEN
S
C0AC  A9A0      1020 PRTNOR LDA #$A0
C0AE  A29E      1030 LDX #$9E
C0B0  8D32A7    1040 LBL000 STA $A732
C0B3  8E31A7    1050 STX $A731
C0B6  8D3AA7    1060 STA $A73A
C0B9  8E39A7    1070 STX $A739
C0BC  20EBC0    1080 JSR GETREG
C0BF  4C1AA7    1090 JMP $A71A
C0C2            1100 ;---------------------
-----------------
C0C2            1110 ; CRUNCH TOKENS ROUT
INE EXTRA CODE
C0C2            1120 ; USE BY ALTERING $A
604 TO 'JMP CRUNCH'
C0C2  20FAC0    1130 CRUNCH JSR PUTREG
C0C5  ADFCA5    1140 LDA $A5FC
C0C8  C9A0      1150 CMP #$A0
C0CA  D00F      1160 BNE STAND
C0CC  A9C0      1170 LDA #$C0
C0CE  A200      1180 LDX #$00
C0D0  200CC1    1190 JSR TOKSTR
C0D3  20EBC0    1200 JSR GETREG
C0D6  A000      1210 LDY #0
C0D8  4CB8A5    1220 JMP $A5B8
C0DB  A9A0      1230 STAND LDA #$A0
C0DD  A29E      1240 LDX #$9E
C0DF  200CC1    1250 JSR TOKSTR
C0E2  20EBC0    1260 JSR GETREG
C0E5  BD0002    1270 LDA $200,X
C0E8  4C07A6    1280 JMP $A607
C0EB            1290 ;---------------------
-----------------
C0EB  AD28C1    1300 GETREG LDA PSAVE
C0EE  48        1310 PHA
C0EF  AD29C1    1320 LDA ASAVE
C0F2  AE2AC1    1330 LDX XSAVE
C0F5  AC2BC1    1340 LDY YSAVE
C0F8  28        1350 PLP
C0F9  60        1360 RTS
C0FA            1370 ;---------------------
-----------------
C0FA  08        1380 PUTREG PHP
C0FB  8D29C1    1390 STA ASAVE
```

```
C0FE  8E2AC1    1400 STX XSAVE
C101  8C2BC1    1410 STY YSAVE
C104  68        1420 PLA
C105  8D28C1    1430 STA PSAVE
C108  AD29C1    1440 LDA ASAVE
C10B  60        1450 RTS
C10C            1460 ;-------------------
---------------------
C10C  8DBEA5    1470 TOKSTR STA $A5BE
C10F  D8        1480 CLD
C110  8EBDA5    1490 STX $A5BD
C113  8D01A6    1500 STA $A601
C116  8E00A6    1510 STX $A600
C119  CA        1520 DEX
C11A  E0FF      1530 CPX #$FF
C11C  D003      1540 BNE LBL003
C11E  38        1550 SEC
C11F  E901      1560 SBC #1
C121  8DFCA5    1570 LBL003 STA $A5FC
C124  8EFBA5    1580 STX $A5FB
C127  60        1590 RTS
C128            1600 ;-------------------
---------------------
C128            1610 PSAVE
C128            1620 ASAVE = PSAVE+1
C128            1630 XSAVE = ASAVE+1
C128            1640 YSAVE = XSAVE+1

TOTAL ERRORS IN FILE --- 0



KEYWRD          C000
ACTVEC          C04F
NORMAL          4B
NEWACT          D
NEWFUN          3
EXECUT          C069
DOEX            C072
LABEL           C08F
DOLET           C090
RETURN          C093
PRTTOK          C096
PRTNOR          C0AC
LBL000          C0B0
```

```
CRUNCH          C0C2
STAND           C0DB
GETREG          C0EB
PUTREG          C0FA
TOKSTR          C10C
LBL003          C121
PSAVE           C128
ASAVE           C129
XSAVE           C12A
YSAVE           C12B
 total number of symbols --- 23
```

CAPITOLO 7
BASIC EXTENDER II

Abbiamo già introdotto un breve programma che trasferisce l'interprete dalla ROM alla RAM, permettendo così di fare cambiamenti. Ora che sono state discusse alcune delle necessarie aggiunte al codice macchina, siamo in grado di aggiornare quel programma BASIC in modo da poter iniziare ad estendere il BASIC. Il listing completo di questo programma esteso viene ora presentato, sebbene molte linee siano identiche a quelle della versione che avete già — commenteremo solo le variazioni.

BASIC extender II: listing

```
0 REM12345678901234567890123456789012345
67890123456789012345678901234567890123
1 REM12345678901234567890123456789012345
6789012345678901234567890123456789012345678900
2 REM12345678901234567890123456789012345
67890123456789012345678901234567890
3 REM12345678901234567890123456789012345
67890123456789012345678901234567890
20 DEV = 1
100 REM BASIC EXTENDER ROUTINE
110 REM MOVE BASIC ROM INTO RAM AT $A000
 - $BFFF
120 DATA 165,1,41,254,133,1,96,160,255,2
00,32,32,8,190,1,1,32,6,8,138,153,1,1
130 DATA 200,208,240,165,1,9,1,133,1,96,
32,6,8,24,162,255,232,160,255,200,185
```

```
135 DATA 75,8,133,20,185,151,8,48,01,96,
133,21,185,227,8,129,20,144,235
139 DATA 0
140 AD = 2054
150 READ A : IF A<>0 THEN POKE AD,A : AD
 = AD+1 : GOTO 150
155 REM LOAD MACHINE CODE FROM TAPE/DISK
156 INPUT " FILE NAME "; IN$ : IF DEV=8
THEN IN$ = IN$+",S,R"
157 OPEN 2,DEV,0,IN$ : INPUT# 2,SA,EA :
FOR X = SA TO EA : INPUT# 2,T : POKE X,T
158 NEXT X : CLOSE 2
160 REM DO ACTUAL MOVE
165 POKE 2068,0 : POKE 2075,0
170 FOR X = 160 TO 191
190 POKE 2069,X : POKE 2076,X
200 SYS 2061
210 NEXT
220 POKE 2068,1 : POKE 2075,1
221 REM DATA FOR ROM EXCUTE ALTERATION
223 X = 0 : DATA 225,167,76,226,167,105,
227,167,192
227 DATA 0
228 READ T1 : IF T1 = 0 THEN 230
229 READ T2,T3 : POKE 2123+X,T1:POKE 219
9+X,T2:POKE 2275+X,T3:X=X+1:GOTO 228
230 SYS 2087 : REM ALTER ROM
231 REM ALTER CRUNCH TOKENS ROUTINE
232 POKE 42500,76 : POKE 42501,194 : POK
E 42502,192
241 POKE 774,150 : POKE 775,192
300 END
```

Commento

1-3: le frasi DATA del programma originale provvedono ad una terza routine macchina che non è stata spiegata. Queste frasi REM fanno spazio per i dati su cui lavorerà questa terza routine.

20: con questo programma esteso caricheremo un file di codice macchina. Il dispositivo indicato è il registratore a cassette, così se lavorate col disco dovrete cambiarlo col numero 8.

155-158: queste linee caricano un file in linguaggio macchina dal dispositivo specificato – l'estensore in linguaggio macchina che abbiamo appena descritto. Le linee equivalgono al caricatore di codice macchina del monitor.

223-227: dati per la terza routine macchina che inizia al decimo elemento della linea 130 DATA. La terza routine sarà usata per immettere con un POKE nella routine di esecuzione dell'interprete un nuovo indirizzo che provocherà un salto alla routine d'esecuzione modificata nell'estensore in linguaggio macchina. Questo va fatto in codice macchina perché si devono modificare due byte. Modificare un solo byte del salto con un POKE da BASIC avrebbe prodotto un salto non corretto al momento dell'interpretazione del successivo POKE. I dati nella linea 223 specificano tre indirizzi con i valori per il byte basso, il byte alto ed il numero che vi andrà immesso. Il listing assembler per la terza routine è il seguente:

MODIFICATORE DELLA ROUTINE DI ESECUZIONE:
listing in linguaggio assemblatore

```
827  200608   JSR $0806
82A  18       CLC
82B  A2FF     LDX #$FF
82D  E8       INX
82E  A0FF     LDY #$FF
830  C8       INY
831  B94B08   LDA $084B,Y
834  8514     STA $14
836  B99708   LDA $0897,Y
839  3001     BMI $83C
83B  60       RTS
83C  8515     STA $15
83E  B9E308   LDA $08E3,Y
841  8114     STA ($14,X)
843  90EB     BCC $830

CONTINUE ( Y/N ) :
```

Commento sul codice macchina

Questa routine prende indirizzi a due byte dai dati nelle frasi REM e vi pone nuove informazioni (prese anch'esse dalle frasi REM). Le istruzioni agli indirizzi 827-82E 'accendono' la RAM ed inizializzano il flag di riporto, i registri X ed Y.

Le linee 830-83C caricano i byte bassi ed alti degli indirizzi dalle frasi REM alla linea 1 del programma BASIC, eseguendo un controllo sui byte alti per vedere se ricadono nell'intervallo corretto di indirizzi per l'interprete. Le linee 83E-843 prelevano e memorizzano i nuovi dati, saltando indietro per ricevere nuovi indirizzi.

Torniamo al programma BASIC:

228-229: questi nuovi dati vengono inseriti via POKE nelle linee 1,2,3 per essere prelevati dalla terza routine macchina in linea 0.

230: ora diviene SYS 2087 invece di SYS 2054. La routine macchina in 2054 viene ancora chiamata, ma dall'interno della terza routine macchina alla linea 0. I cambiamenti necessari per l'esecuzione della routine verranno completati da questa chiamata.

231-232: il nuovo indirizzo viene immesso con POKE nella routine di creazione dei token — cosa che si può fare da BASIC perché la routine di creazione token non è usata durante l'esecuzione di un programma (non si potrebbe fare in modo diretto dal momento che i token dovrebbero venir creati in esecuzione).

240-241 viene modificato il vettore 'stampa token' per puntare alla nostra routine modificata.

Una volta modificato il vostro primo programma BASIC extender, o immessa ex novo questa versione, la cosa migliore sarebbe salvare questo programma all'inizio di un nastro, lasciare uno spazio per ulteriori estensioni al programma e quindi salvare sullo stesso nastro la versione assemblata del programma visto al capitolo 6. Quando il programma verrà eseguito, vi verrà chiesto di indicare il nome del file e voi dovreste rispondere con qualsiasi nome abbiate dato al programma del capitolo 6 quando l'avete salvato. Vi basta premere 'PLAY' ora per caricarlo, evitando così un gran lavoro di scambio di cassette.

Una volta assemblatolo ed introdottolo in memoria con questa tecnica, dovreste essere in grado di immettere qualsiasi nuova parola chiave azione (non ancora funzione) specificata nella tabella. Come detto nessuna di esse farà ancora nulla se non eseguire END, ma presto le cose cambieranno!

CAPITOLO 8
PARULE CHIAVE BASIC PER AZIONI

SEZIONE 1: UNDEAD

Ora che sono state presentate le routine BASIC oltre al codice macchina per estendere il BASIC, è tempo di introdurre le routine che permettono alle nuove parole chiave di fare qualcosa. In questa sezione del libro discuteremo tre parole chiave per azioni che sono più semplici di altre, non richiedendo l'introduzione di parametri per la loro esecuzione. Nel normale BASIC, ad esempio, il comando GOTO 10 non richiede solo una routine per eseguire il GOTO, ma anche una che prelevi il parametro 10, senza il quale il GOTO sarebbe privo di senso. D'altra parte STOP non richiede parametri oltre alla parola chiave stessa. Più avanti mostreremo come aggiungere parole chiave che richiedono parametri.

La parola chiave che aggiungeremo ora è UNDEAD – alcuni la chiamano 'OLD', ma la notte precedente quella in cui venne scritta questa routine la TV trasmetteva un film di vampiri...

L'effetto del comando è quello di superare quel seccante problema che di tanto in tanto tutti devono affrontare – si batte NEW per togliere un programma dalla memoria ed improvvisamente ci si ricorda di non averlo salvato. UNDEAD ripristinerà il programma con la sola eccezione che le variabili andranno perdute.

UNDEAD: listing in linguaggio assemblatore

```
10        PRT
20        SYM
30        ORG $C05D
40        WRD UNDEAD-1
50        ORG $C1E3
60        UNDEAD LDA #$FF
```

```
 70           LDY #1
 80           STA ($2B).Y
 90           JSR $A533
 100           LDA $22
 110           CLC
 120           CLD
 130           ADC #2
 140           STA $2D
 150           LDA $23
 160           ADC #0
 170           STA $2E
 180           JMP $A65E
 190           END

END
```

30-40: ricorderete da qualche pagina più addietro che tutti i vettori azione per le nuove parole chiave puntano al momento a END. Queste due linee pongono l'indirizzo dell'etichetta UNDEAD nell'appropriato vettore azioni della nuova tabella.

50: le routine macchina delle nuove parole chiave occuperanno, insieme all'extender macchina, un'area di memoria fra C000 e C4B5 esadecimale. Non verranno immesse in ordine, ma la cosa non avrà conseguenze sulla loro esecuzione.

60-90: quando si esegue NEW, una delle modifiche che essa fa è costituita dal porre tre BYTES contenenti zero all'inizio dell'area di programma (800-802 esadecimale), sebbene uno di essi (800) sia comunque sempre uguale a zero. Tre zeri rappresentano l'indicatore di fine programma per l'interprete, quindi ogni tentativo di eseguire o listare il programma dopo NEW termina prima di iniziare, nonostante il programma sia intatto in memoria. Queste linee pongono il valore esadecimale FF nel terzo byte dell'area di programma, essendo questo il byte alto dei due byte di collegamento della prima riga di programma BASIC — il valore nei due byte è ora senza significato, ma non ha importanza. Viene ora chiamata la routine 'ricollega linee' dell'interprete, all'indirizzo A533 esadecimale, che esamina il programma ripristinando tutti i byte di collegamento che iniziano ogni linea BASIC e puntano all'inizio della seguente.

100-170: la routine di ricollegamento delle linee usa i due byte di memoria posti alle locazioni 22-23 esadecimali come una variabile che indichi il punto cui si è arrivati nel programma e, quando l'elaborazione è terminata, questi byte puntano al termine dell'ultima linea del programma BASIC. Questo valore viene prelevato e posto

nel puntatore principale che registra la fine del programma BASIC, con l'aggiunta di 2 per tener conto dei due zeri extra che indicano la vera fine del programma.

180: ci sono vari altri puntatori da ripristinare prima di recuperare con successo il programma, ma ciò si può fare con una semplice chiamata alla routine CLR dell'interprete normale.

190: notate l'uso della direttiva END. È utile non solo per indicare il termine del listing in linguaggio assemblatore all'assembler Mastercode, ma il suo indirizzo è anche il primo byte libero dopo la fine della routine che esso termina, in modo che ogni routine che vada aggiunta dopo questa, possa iniziare all'indirizzo specificato per END nel listing in linguaggio assemblatore.

UNDEAD: listing completamente assemblato

```
add.   data      source code
0                10 PRT
0                20 SYM
0                30 ORG $C05D
C05D   E2C1      40 WRD UNDEAD-1
C05F             50 ORG $C1E3
C1E3   A9FF      60 UNDEAD LDA #$FF
C1E5   A001      70 LDY #1
C1E7   912B      80 STA ($2B).Y
C1E9   2033A5    90 JSR $A533
C1EC   A522      100 LDA $22
C1EE   18        110 CLC
C1EF   D8        120 CLD
C1F0   6902      130 ADC #2
C1F2   852D      140 STA $2D
C1F4   A523      150 LDA $23
C1F6   6900      160 ADC #0
C1F8   852E      170 STA $2E
C1FA   4C5EA6    180 JMP $A65E
C1FD             190 END

 TOTAL ERRORS IN FILE --- 0


UNDEAD              C1E3
 total number of symbols --- 1
```

La procedura da seguire per immettere questa routine è:

1) caricate il programma Mastercode e per prima cosa richiamate il file in codice macchina dell'extender che avete introdotto in passato. La ragione per cui questo è necessario è che nell'assemblaggio di UNDEAD uno dei vettori azione della nuova tabella viene riscritto – questo non otterrà molto se il programma predetto manca!

2) immettete il listing in linguaggio assembler di UNDEAD usando il file editor.

3) richiamate l'assembler ed assemblate UNDEAD.

4) salvate l'area di memoria compresa fra C000 e C1FC nella forma di file in codice macchina o, meglio ancora, l'area fra C000 e C4B5, che è l'intera zona che sarà poi usata dalle nuove routine. Usando questo secondo metodo ogni volta che si immette una nuova routine, sarete sicuri di non tagliare inavvertitamente alcuna delle routine seguenti come posizione in memoria, ma immesse precedentemente a quella corrente.

5) caricate l'estensore BASIC II e fatelo eseguire, dandogli il nome del file contenente l'estensore ed UNDEAD.

6) se tutto ha funzionato a dovere, potete ora dare NEW e poi cercare di dare LIST – non apparirà nulla, come vi aspettate.

7) immettete in modo diretto UNDEAD e premete RETURN. Date ancora LIST e dovreste vedere il programma completamente riportato in vita.

UNDEAD: note sull'uso

UNDEAD può ripristinare un programma solo se non si è scritto nulla su di esso in memoria. Se dopo aver dato NEW immettete una linea BASIC o dichiarate una variabile, non c'è modo in cui UNDEAD vi possa aiutare, poichè il programma non è più intatto in memoria.

SEZIONE 2: Subex

Immettete il seguente programma nel vostro 64:

```
10 GOSUB 20
20 GOTO 10
```

Ora fatelo eseguire e vedrete in un istante che avete esaurito la memoria-come? Ogni volta che una subroutine viene chiamata, l'indirizzo cui deve saltare l'esecuzione quando si incontra RETURN viene memorizzato in un'area detta 'pila di ritorno'. Ogni RETURN prende l'ultimo indirizzo della pila e vi salta, ogni GOSUB aggiunge un indirizzo sopra quello già presente in cima alla pila. Uscendo da una subroutine con un mezzo diverso dal RETURN, l'indirizzo di ritorno resta sulla pila. Se la cosa si ripete spesso la pila si riempie e si genera un errore di tipo 'OUT OF MEMORY'.

Ammettiamo che uscire da una subroutine senza un RETURN non è una pratica da incoraggiare troppo e, dopo tutto, perché usare un GOSUB se non si vuole tornare in quel punto? Ci sono però circostanze nelle quali può essere estremamente utile uscire da una subroutine senza ingombrare la memoria. Se guardate il linguaggio assemblatore, ad esempio, vedrete che ci sono molti casi di subroutine che chiamano subroutine che chiamano subroutine... Non c'è nulla di male in questo, anzi è buona pratica di programmazione mettere il maggior numero di subroutine possibile. Ma cosa succede se, a distanza di quattro o cinque subroutine dalla routine di controllo trovate una condizione che comporta la non esecuzione della catena di subroutine — ad esempio un errore nei dati in ingresso? Ciò che desiderate è segnalare l'errore e tornare immediatamente alla routine di controllo in modo da prendere i necessari provvedimenti, ma ciò non è possibile nel BASIC standard.

Dovete risalire attraverso le subroutine, inserendovi ogni volta una linea che rilevi la segnalazione d'errore e ritorni ancora fino a raggiungere la routine di controllo. In un programma complesso ciò può implicare molte linee in più ed un considerevole costo in termini di tempo.

La risposta al problema è il comando SUBEX. Tutto ciò che fa è rimuovere l'ultimo indirizzo di ritorno dalla pila, in modo che possiate uscire dall'ultima subroutine senza ingombrare la pila. Ovviamente se volete saltare direttamente dalla quinta subroutine di una catena all'inizio della stessa SUBEX dovrà venir eseguito cinque volte, ma il risparmio di tempo e di complessità di programma può essere spesso considerevole. Con un SUBEX un programma del tipo:

```
10 GOSUB 20
20 SUBEX
30 GOTO 10
```

proseguirà indefinitamente.

SUBEX: listing in linguaggio assemblatore

```
 10         PRT
 20         SYM
 30         ORG $C05F
 40         WRD SUBEX-1
 50         ORG $C1FD
 60         SUBEX LDA #$FF
 70         STA $4A
 80         JSR $A38A
 90         TXS
 100         CMP #$8D
 110         BNE RETERR
 120         PLA
 130         PLA
 140         PLA
 150         PLA
 160         PLA
 170         RTS
 180         RETERR JMP $A8E0
 190         END

END
```

60-80: quando si esegue un RETURN nel BASIC normale viene chiamata la routine alla posizione A38A esadecimale per trovare il primo indirizzo di ritorno sulla pila, cosa necessaria perché potrebbero essere in pila anche dati necessari per un ciclo FOR. La routine a quell'indirizzo manipola anche la pila in modo che il primo indirizzo di ritorno sia posto in cima alla pila. Queste linee pongono l'indirizzo di memoria uguale a 4A esadecimali, il che inizializza la routine di ricerca, poi la routine viene chiamata.

90-110: al ritorno dalla routine di ricerca in pila, l'ultimo valore da porre in pila sarà, si spera, un puntatore alla posizione dell'indirizzo di ritorno nella pila ed esso viene immagazzinato nel registro X. Questo è confermato inserendo il valore 8D in accumulatore. Se non è stato trovato un indirizzo di ritorno l'accumulatore non conterrà 8D e verrà fatto un salto a RETERR, che specifica un salto al messaggio di errore 'RETURN WITHOUT GOSUB ERROR'.

120-170: i cinque byte dell'indirizzo di ritorno vengono tolti dalla pila ed eliminati. La routine SUBEX è completata e ritorna.

SUBEX: listing completamente assemblato

```
add.   data     source code
0               10 PRT
0               20 SYM
0               30 ORG $C05F
C05F  FCC1      40 WRD SUBEX-1
C061            50 ORG $C1FD
C1FD  A9FF      60 SUBEX LDA #$FF
C1FF  854A      70 STA $4A
C201  208AA3    80 JSR $A38A
C204  9A        90 TXS
C205  C98D      100 CMP #$8D
C207  D006      110 BNE RETERR
C209  68        120 PLA
C20A  68        130 PLA
C20B  68        140 PLA
C20C  68        150 PLA
C20D  68        160 PLA
C20E  60        170 RTS
C20F  4CE0A8    180 RETERR JMP $A8E0
C212            190 END

 TOTAL ERRORS IN FILE --- 0


SUBEX            C1FD
RETERR           C20F
 total number of symbols --- 2
```

Come per UNDEAD, la procedura per caricare SUBEX in memoria è di chiamare prima l'ultimo file in codice macchina che avete salvato (extender-UNDEAD), caricatelo in memoria poi assemblate SUBEX. Salvate l'intera area di memoria e fate eseguire poi il BASIC extender II, dando il nome del vostro nuovo file. Dovreste ora essere in grado di far eseguire il secondo programma BASIC dato nell'introduzione di SUBEX senza terminare la memoria.

SUBEX: note sull'uso

SUBEX è un comando potente ma c'è bisogno di attenzione nell'uso. Se siete all'interno di una catena di subroutine, è essenziale che contiate esattamente quanti RETURN desiderate saltare o vi capiterà di rientrare nella routine sbagliata o di ricevere un errore di tipo 'RETURN WITHOUT GOSUB'.

SEZIONE 3: RKILL

Durante lo sviluppo di un programma è buona norma includere tante frasi REM quante sono necessarie ad assicurarvi di comprendere il programma la prossima volta che vi lavorerete. La leggibilità inoltre è migliorata spaziando bene i comandi sulle linee. Quando il programma è terminato però i REM e gli spazi extra sono semplicemente memoria sprecata che potrebbe essere meglio impiegata. RKILL risolve questo problema eliminando tutti i REM posti al termine di linee BASIC e tutti gli spazi fuori dalle virgolette. Notate che non rimuove i REM che occupano da soli una linea poiché, se scrivete correttamente i vostri programmi, potrebbero essere le intestazioni delle diverse sezioni del programma. I GOTO e GOSUB punteranno ad essi, permettendo così di aggiungere nuove linee iniziali alle sezioni senza dover modificare i numerosi GOTO e GOSUB nel programma.

BKILL: listing completamente assemblato

```
 10         PRT
 20         SYM
 30         QFLAG = $F
 40         REMTOK = $8F
 50         ASAVE = $C105
 60         XSAVE = $C106
 70         ORG $C051
 80         WRD RKILL-1
 90         ORG $C160
 100         RKILL LDA #$FF
 110         STA $14
 120         STA $15
 130         ; SAVE PRESENT BASIC WARM STA
RT LINK
 140         LDA $302
 150         LDX $303
 160         STA TEMP
 170         STX TEMP+1
 180         ; PUT NEW WARM START LINK IN
 190         LDA #LBL003-LBL003/256*256
 200         LDX #LBL003/256
 210         STA $302
 220         STX $303
 230         ; GET NEXT LINE NO TO BE TREA
TED
```

```
 240         LBL003 INC $14
 250         BNE LBL004
 260         INC $15
 270         ; USE ROM ROUTINE TO GET ADD
OF LINE IN $5F & $60
 280         LBL004 JSR $A613
 290         ; IF HI LINK BYTE = 0 THEN EX
IT
 300         LDY #1
 310         LDA ($5F).Y
 320         BEQ LBL009
 330         ; GET THIS LINE NO. INTO $14
& $15
 340         INY
 350         LDA ($5F).Y
 360         STA $14
 370         INY
 380         LDA ($5F).Y
 390         STA $15
 400         ; COPY LINE TO INPUT BUFFER D
ELETING SPACES-EXCEPT IN QUOTES
 410         LDX #4
 420         STX QFLAG
 430         LBL005 INY
 440         LDA ($5F).Y
 450         ;IF BYTE = 0 THIS IS THE END
OF THE LINE SO INPUT IT
 460         BEQ LBL007
 470         ; IF ITS A QUOTE THEN TOGGLE
THE QUOTES FLAG
 480         CMP #34
 490         BNE LBL006
 500         LDA QFLAG
 510         EOR #$FF
 520         STA QFLAG
 530         LDA #34
 540         ;IF THE QUOTFLAG IS SET DONT
DELETE ANYTHING
 550         LBL006 BIT QFLAG
 560         BMI LBL008
 570         ; TEST FOR SPACE & DELETE IT
IF FOUND
 580         CMP #$20
 590         BEQ LBL005
 600         ;TRANSFER FIRST NON-SPACE EVE
N IF ITS A REM
 610         CMP #REMTOK
```

```
620        BNE LBL008
630        CPX #4
640        BNE LBL002
650        INX
660        STA $1FB.X
670        INX
680        LBL002 DEX
690        LBL007 LDA #0
700        INX
710        STA $1FB.X
720        STX $B
730        JMP $A4A4
740        LBL008 INX
750        STA $1FB.X
760        JMP LBL005
770        LBL009 LDA TEMP
780        LDX TEMP+1
790        STA $302
800        STX $303
810        JMP $A474
820        TEMP WRD 0
830        END

END
```

100-120: le locazioni 14-15 sono usate dal file editor BASIC quando per qualche motivo lavora entro un programma. Essi vengono caricati con 255 ciascuno, in modo che all'inizio della routine principale l'aggiunta di uno li riporti a zero.

130-170: durante questa istruzione chiameremo la routine di ingresso linee, che elabora una linea posta da tastiera nel buffer di input. Quando questo processo termina, la routine torna normalmente ad uno stato d'attesa di input da tastiera. Noi vogliamo che torni alla routine RKILL, quindi salviamo l'indirizzo di collegamento esistente in 302-303 (esadecimali) in modo da potervi piazzare un collegamento a questa routine.

180-220: viene installato un nuovo indirizzo di collegamento che punta a questa routine.

230-260: viene incrementato di uno il numero di linea su cui si sta lavorando.

270-280: viene usata una routine ROM per trovare l'indirizzo di quella linea in memoria. Se non c'è una linea del genere, la routine restituirà l'indirizzo della linea col numero seguente più vicino.

290-320: se il byte alto del byte di collegamento all'indirizzo indicato dalla routine ROM è zero, allora è stata trovata la fine del file e la routine termina.

330-390: il numero della linea trovata viene prelevato e memorizzato alle locazioni 14-15 esadecimali.

410-420: queste linee inizializzano la routine. X sarà l'inizio del testo della linea nel buffer d'ingresso (ricavato dai byte di collegamento e dal numero di linea); il segnale di virgolette, che registra se un carattere in ingresso è fra virgolette, viene inizializzato.

430-440: viene posto in accumulatore il prossimo byte della linea.

450-460: se il byte è zero, è stata raggiunta la fine linea e viene fatto un salto alla routine ROM che introduce effettivamente una linea.

470-530: se il carattere prelevato è un punto di domanda, il segnale di virgolette è posto uguale a zero o uno a seconda che sia il primo o il secondo segno di virgolette di una coppia.

550-560: se il segnale di virgolette è a zero, la parte principale della routine viene tralasciata poiché non vogliamo interferire col contenuto delle righe fra virgolette.

570-590: se il carattere prelevato è uno spazio, viene ignorato e si passa al successivo carattere.

610-620: si fa un controllo sul token di REM: se il carattere non corrisponde a REM, viene posto nel buffer d'ingresso.

630-680: se è stato rilevato un token di REM, si fa un test per vedere se si trova a inizio riga (il registro X è in questo caso 4). Se è così, viene posto nel file delle linee 650-660. Altrimenti il registro X viene decrementato per puntare ai due punti precedenti nel buffer d'ingresso. Il puntatore contenuto nel registro X punterà ora ai due punti precedenti il REM se il REM non si trovava a inizio linea ed alla posizione seguente il REM in caso contrario (tutti questi si trovano comunque, ricordate, nel buffer e non nella linea vera e propria).

690-730: viene ora memorizzato uno zero nel buffer d'ingresso segnalando così la fine riga. La lunghezza della linea viene piazzata nella locazione B (esadecimale) e viene poi eseguita la routine di input, ponendo così la linea accorciata nel programma invece dell'originale.

740-760: a questo punto tutti i test precedenti hanno fallito, così il carattere non dev'essere cancellato e dobbiamo memorizzarlo nel buffer d'ingresso per poi ripetere il ciclo e prelevare il carattere seguente.

770-810: è la routine d'uscita da RKILL, quindi viene ripristinato il normale collegamento con la tastiera e si salta alla routine del modo diretto, provocando così il messaggio 'READY'.

820: TEMP stabilisce la locazione di memoria per il collegamento originale al 'warmstart' (partenza a caldo o reinizializzazione del sistema), che viene salvato all'inizio di RKILL.

RKILL: listing completamente assemblato

```
add.   data      source code
0                10 PRT
0                20 SYM
0                30 QFLAG = $F
0                40 REMTOK = $8F
0                50 ASAVE = $C105
0                60 XSAVE = $C106
0                70 ORG $C051
C051   5FC1      80 WRD RKILL-1
C053             90 ORG $C160
C160   A9FF      100 RKILL LDA #$FF
C162   8514      110 STA $14
C164   8515      120 STA $15
C166             130 ; SAVE PRESENT BASIC
WARM START LINK
C166   AD0203    140 LDA $302
C169   AE0303    150 LDX $303
C16C   8DE1C1    160 STA TEMP
C16F   8EE2C1    170 STX TEMP+1
C172             180 ; PUT NEW WARM START
LINK IN
```

```
C172  A97C     190 LDA #LBL003-LBL003/25
6*256
C174  A2C1     200 LDX #LBL003/256
C176  8D0203   210 STA $302
C179  8E0303   220 STX $303
C17C           230 ; GET NEXT LINE NO TO
 BE TREATED
C17C  E614     240 LBL003 INC $14
C17E  D002     250 BNE LBL004
C180  E615     260 INC $15
C182           270 ; USE ROM ROUTINE TO
GET ADD OF LINE IN $5F & $60
C182  2013A6   280 LBL004 JSR $A613
C185           290 ; IF HI LINK BYTE = 0
 THEN EXIT
C185  A001     300 LDY #1
C187  B15F     310 LDA ($5F).Y
C189  F047     320 BEQ LBL009
C18B           330 ; GET THIS LINE NO. I
NTO $14 & $15
C18B  C8       340 INY
C18C  B15F     350 LDA ($5F).Y
C190  C8       370 INY
C191  B15F     380 LDA ($5F).Y
C193  8515     390 STA $15
C195           400 ; COPY LINE TO INPUT
BUFFER DELETING SPACES-EXCEPT IN QUOTES
C195  A204     410 LDX #4
C197  860F     420 STX QFLAG
C199  C8       430 LBL005 INY
C19A  B15F     440 LDA ($5F).Y
C19C           450 ;IF BYTE = 0 THIS IS
THE END OF THE LINE SO INPUT IT
C19C  F022     460 BEQ LBL007
C19E           470 ; IF ITS A QUOTE THEN
 TOGGLE THE QUOTES FLAG
C19E  C922     480 CMP #34
C1A0  D008     490 BNE LBL006
C1A2  A50F     500 LDA QFLAG
C1A4  49FF     510 EOR #$FF
C1A6  850F     520 STA QFLAG
C1A8  A922     530 LDA #34
C1AA           540 ;IF THE QUOTFLAG IS S
ET DONT DELETE ANYTHING
C1AA  240F     550 LBL006 BIT QFLAG
C1AC  301D     560 BMI LBL008
C1AE           570 ; TEST FOR SPACE & DE
LETE IT IF FOUND
```

```
C1AE  C920      580 CMP #$20
C1B0  F0E7      590 BEQ LBL005
C1B2            600 ;TRANSFER FIRST NON-S
PACE EVEN IF ITS A REM
C1B2  C98F      610 CMP #REMTOK
C1B4  D015      620 BNE LBL008
C1B6  E004      630 CPX #4
C1B8  D005      640 BNE LBL002
C1BA  E8        650 INX
C1BB  9DFB01    660 STA $1FB.X
C1BE  E8        670 INX
C1BF  CA        680 LBL002 DEX
C1C0  A900      690 LBL007 LDA #0
C1C2  E8        700 INX
C1C3  9DFB01    710 STA $1FB.X
C1C6  860B      720 STX $B
C1C8  4CA4A4    730 JMP $A4A4
C1CB  E8        740 LBL008 INX
C1CC  9DFB01    750 STA $1FB.X
C1CF  4C99C1    760 JMP LBL005
C1D2  ADE1C1    770 LBL009 LDA TEMP
C1D5  AEE2C1    780 LDX TEMP+1
C1D8  8D0203    790 STA $302
C1DB  8E0303    800 STX $303
C1DE  4C74A4    810 JMP $A474
C1E1  0000      820 TEMP WRD 0
C1E3            830 END

 TOTAL ERRORS IN FILE --- 0


QFLAG           F
REMTOK          8F
ASAVE           C105
XSAVE           C106
RKILL           C160
LBL003          C17C
LBL004          C182
LBL005          C199
LBL006          C1AA
LBL002          C1BF
LBL007          C1C0
LBL008          C1CB
LBL009          C1D2
TEMP            C1E1

 total number of symbols --- 14
```

Come per la routine precedente, per venir introdotto RKILL dev'essere prima assemblato nel vostro file generale che, a questo punto, dovrebbe consistere dell'extender, di UNDEAD e di SUBEX. Il file dovrebbe venir salvato e poi posto in memoria con il BASIC extender II. Una volta caricato il file, battere direttamente RKILL dovrebbe eliminare spazi e REM dal BASIC extender.

RKILL: note sull'uso

RKILL è un comando in modo diretto – non può venir inserito in una linea di programma. La ragione di ciò è che, poiché abbrevia il programma, altera i puntatori usati dall'interprete. Fare questo mentre un programma è in esecuzione è una ricetta sicura per combinare un pasticcio, perciò RKILL termina l'esecuzione di un programma una volta completato (come LIST).

CAPITOLO 9
IL PROBLEMA DEI PARAMETRI

SEZIONE 1: GETWRD

Abbiamo già notato che alcune parole chiave per azioni richiedono che vengano prelevate dalla linea di programma altre informazioni prima che essa possa venir eseguita. Questa routine dall'aspetto insignificante realizza questa funzione per le nostre nuove parole chiave e dev'essere immessa nel file globale di codice macchina prima che noi possiamo introdurre nuove parole chiave con parametri. Il suo nome è GETWRD.

GETWRD: listing in linguaggio assemblatore

```
 10        PRT
 20        SYM
 30        ORG $C12C
 40        ; ROUTINE TO GET 16 BIT UNSIGN
ED INTEGER FROM BASIC INTO $14 & $15
 50        GETWRD JSR $AD8A
 60        JMP $B7F7
 70        END

END
```

50: la routine qui chiamata è nell'interprete standard e si limita semplicemente a prelevare un numero in virgola mobile dalla linea BASIC attualmente in elaborazione. I numeri immessi in modo non corretto provocheranno un errore sintattico, come per una normale parola chiave BASIC.

60: questa routine trasforma il numero prelevato in un intero dell'intervallo 0-65535. Numeri esterni a quest'intervallo provocano un errore del tipo 'ILLEGAL QUANTITY' (quantità non lecita).

GETWRD: listing completamente assemblato

```
add.   data      source code
0                10 PRT
0                20 SYM
0                30 ORG $C12C
C12C             40 ; ROUTINE TO GET 16 BI
T UNSUGNED INTEGER FROM BASIC INTO $14 &
 $15
C12C   208AAD    50 GETWRD JSR $AD8A
C12F   4CF7B7    60 JMP $B7F7
C132             70 END

 TOTAL ERRORS IN FILE --- 0


GETWRD            C12C
 total number of symbols --- 1
```

Sì, questo è tutto ciò che serve. Non potete ancora fare nulla con la routine, ma i prossimi comandi la useranno intensamente per restituire i parametri per le nostre nuove parole chiave. Come le routine precedenti, dev'essere immessa nel codice macchina globale prima di passare alla sezione seguente.

SEZIONE 2: DOKE

Ora che possiamo prelevare parametri per nuove parole chiave, tutta una serie di nuove possibilità si apre. La prima di queste è DOKE, che semplicemente pone in memoria un numero compreso fra 0 e 65535 in una locazione a due byte. Questo evita tutte le complicazioni di immettere espressioni come VAR - 256 INT(VAR/256) ogni volta che si devono mettere in memoria numeri a due byte.

DOKE: listing in linguaggio assemblatore

```
 10        PRT
 20        GETWRD = $C12C
 30        SYM
 40        ORG $C04F
 50        WRD DOKE-1
 60        ORG $C212
 70        DOKE JSR GETWRD
 80        ; CHECK FOR A COMMA
 90        JSR $AEFD
 100        ; PUT ADDRESS ON STACK WHILE
GETTING THE DATA
 110        LDA $14
 120        PHA
 130        LDA $15
 140        PHA
 150        ; GET VALUE TO BE DOKED
 160        JSR GETWRD
 170        ; PUT ADDRESS INTO TEMPORARY
POINTER
 180        LDX $15
 190        LDY $14
 200        PLA
 210        STA $15
 220        PLA
 230        STA $14
 240        TYA
 250        ; USE ROM ROUTINE TO SET  I TO
 ZERO & PUT FIRST BYTE IN MEMORY
 260        JSR $B828
 270        INY
 280        TXA
 290        STA ($14).Y
 300        RTS
 310        END

END
```

Commento

70: ricava, per mezzo di GETWRD, il numero corrispondente all'indirizzo cui la cifra dev'essere immessa via DOKE.

80-90: se non c'è una virgola dopo il primo parametro, si segnala un errore sintattico.

100-140: il primo parametro viene salvato sulla pila dopo essere stato prelevato dalle locazioni esadecimali 14-15 in cui GETWRD l'aveva posto.

160: viene chiamata ancora GETWRD per trovare il valore da immettere con la DOKE.

180-230: il valore viene posto in X ed Y e l'indirizzo viene rimesso nelle locazioni 14-15 esadecimali.

240-260: il byte basso del dato da memorizzare viene posto in accumulatore, quindi viene richiamata una breve sezione di ROM che mette a zero il registro Y e carica il contenuto dell'accumulatore nella locazione indicata dai due byte che iniziano alla posizione 14. Si sarebbe potuto fare in quattro byte in questa stessa routine, ma dal momento che due istruzioni erano nella ROM con un'istruzione di ritorno dopo di loro, perché non usarle?

270-330: il valore di Y viene incrementato di uno ed il byte alto del valore da memorizzare via DOKE è posto nel byte al di sopra di quello in cui è stato piazzato quello basso.

DOKE: listing completamente assemblato

```
add.  data    source code
0             10 PRT
0             20 GETWRD = $C12C
0             30 SYM
0             40 ORG $C04F
C04F  11C2    50 WRD DOKE-1
C051          60 ORG $C212
C212  202CC1  70 DOKE JSR GETWRD
C215          80 ; CHECK FOR A COMMA
C215  20FDAE  90 JSR $AEFD
C218          100 ; PUT ADDRESS ON STAC
K WHILE GETTING THE DATA
C218  A514    110 LDA $14
C21A  48      120 PHA
C21B  A515    130 LDA $15
C21D  48      140 PHA
C21E          150 ; GET VALUE TO BE DOK
ED
C21E  202CC1  160 JSR GETWRD
```

```
C221                170 ; PUT ADDRESS INTO TE
MPORARY POINTER
C221  A615          180 LDX $15
C223  A414          190 LDY $14
C225  68            200 PLA
C226  8515          210 STA $15
C228  68            220 PLA
C229  8514          230 STA $14
C22B  98            240 TYA
C22C                250 ; USE ROM ROUTINE TO
SET  I TO ZERO & PUT FIRST BYTE IN MEMORY
C22C  2028B8        260 JSR $B828
C22F  C8            270 INY
C230  8A            280 TXA
C231  9114          290 STA ($14).Y
C233  60            300 RTS
C234                310 END

 TOTAL ERRORS IN FILE --- 0


GETWRD               C12C
DOKE                 C212
 total number of symbols --- 2
```

DOKE: note sull'uso

Doke può venir usata per sostituire ogni doppio comando POKE, ma ricordate che se il suo argomento è un valore inferiore a 256, il byte al di sopra di quello specificato nel primo parametro verrà azzerato, quindi DOKE 255 non può sostituire POKE. La sintassi corretta per DOKE è:
DOKE <indirizzo>,<valore>

SEZIONE 3: Plot

Non c'è dubbio che i caratteri di controllo del cursore del Commodore diano una grossa flessibilità al momento di trasferire materiale sullo schermo. Ciononostante vi sono casi in cui sarebbe bello poter scrivere qualcosa nel centro dello

schermo senza specificare una lunga lista di caratteri di controllo o stampare una parte di qualche stringa definita in precedenza di quei caratteri. Il nuovo comando PLOT vi permetterà di muovere il cursore a qualsiasi posizione con un singolo comando.

PLOT: listing in linguaggio macchina.

```
 10         PRT
 20         SYM
 30         GETWRD = $C12C
 40         ORG $C05B
 50         WRD PLOT-1
 60         ORG $C3DA
 70         PLOT JSR GETWRD
 80         JSR $AEFD
 90         LDA $15
 100         BNE IQERR
 110         LDA $14
 120         CMP #25
 130         BCS IQERR
 140         PHA
 150         JSR GETWRD
 160         PLA
 170         TAX
 180         LDA $15
 190         BNE IQERR
 200         LDY $14
 210         CPY #40
 220         BCS IQERR
 230         JMP $FFF0
 240         IQERR JMP $B248
 250         END

END
```

Commento

70-130: queste linee ricavano il parametro per il numero di linea (posizione verticale sullo schermo). Controllate che faccia seguito una virgola e che il numero cada nell'intervallo 0-24.

140-220: il numero di linea viene memorizzato sulla pila e poi viene ricavato il numero di colonna, che sarà controllato per vedere se è nell'intervallo 0-39. Il registro X verrà caricato col numero di riga ed Y con quello di colonna.

230: chiama la routine del KERNAL che stabilisce la posizione del cursore. La routine del KERNAL usa i registri X ed Y per ottenere la posizione esatta.

240 : IQERR è l'indirizzo per stampare il messaggio di 'ILLEGAL QUANTITY'.

PLOT: listing completamente assemblato

```
add.   data      source code
0                10 PRT
0                20 SYM
0                30 GETWRD = $C12C
0                40 ORG $C05B
C05B   D9C3      50 WRD PLOT-1
C05D             60 ORG $C3DA
C3DA   202CC1    70 PLOT JSR GETWRD
C3DD   20FDAE    80 JSR $AEFD
C3E0   A515      90 LDA $15
C3E2   D019      100 BNE IQERR
C3E4   A514      110 LDA $14
C3E6   C919      120 CMP #25
C3E8   B013      130 BCS IQERR
C3EA   48        140 PHA
C3EB   202CC1    150 JSR GETWRD
C3EE   68        160 PLA
C3EF   AA        170 TAX
C3F0   A515      180 LDA $15
C3F2   D009      190 BNE IQERR
C3F4   A414      200 LDY $14
C3F6   C028      210 CPY #40
C3F8   B003      220 BCS IQERR
C3FA   4CF0FF    230 JMP $FFF0
C3FD   4C48B2    240 IQERR JMP $B248
C400             250 END

 TOTAL ERRORS IN FILE --- 0


GETWRD           C12C
PLOT             C3DA
IQERR            C3FD
 total number of symbols --- 3
```

PLOT: note sull'uso

PLOT può venir usato per sostituire la maggior parte delle espressioni che usano stringhe di caratteri di controllo del cursore, sebbene questi rimangano utili per le mosse a posizioni relative rispetto a quella corrente del cursore. PLOT può lavorare su valori come su espressioni, del tipo PLOT X+3,Y/2.

La sintassi corretta per PLOT è:
PLOT<numero di righe>,<numero di colonne>

SEZIONE 4: Delete

Cancellare righe una alla volta può essere noioso, specialmente se si può immettere il comando DELETE per rimuovere in un attimo una serie di linee.

DELETE: listing in linguaggio assemblatore

```
10          PRT
20          ; BLOCK DELETE OF LINES
30          SYM
40          ORG $C053
50          WRD DEL-1
60          ORG $C400
70          GETWRD = $C12C
80          DEL JSR GETWRD
90          ; CONVERT TO ADDRESS
100          JSR $A613
110          BCC ULERR
120          ; SAVE POINTER ON STACK
130          LDA $5F
140          PHA
150          LDA $60
160          PHA
170          ; CHECK THAT A - SIGN FOLLOWS
```

```
 180         LDA #45
 190         JSR $AEFF
 200         ; GET LAST NO. TO BE DELETED
 210         JSR GETWRD
 220         JSR $A613
 230         BCC ULERR
 240         ; GET ADDRESS OF END OF LAST
LINE TO BE DELETED
 250         LDY #1
 260         LDA ($5F).Y
 270         TAX
 280         DEY
 290         LDA ($5F).Y
 300         TAY
 310         ; NOW STORE THESE BYTES IN FI
RST LINE TO BE DELETED
 320         PLA
 330         STA $60
 340         PLA
 350         STA $5F
 360         TYA
 370         LDY #0
 380         STA ($5F).Y
 390         INY
 400         TXA
 410         STA ($5F).Y
 420         ; GET LINE NO. TO BE DELETED
 430         INY
 440         LDA ($5F).Y
 450         STA $14
 460         INY
 470         LDA ($5F).Y
 480         STA $15
 490         ; PUT ZERO INTO BASIC INPUT B
UFFER - TELL FILE ED. TO DELETE LINE
 500         LDA #0
 510         STA $200
 520         ; TIDY UP RETURN STACK
 530         PLA
 540         PLA
 550         ; USE ROM ROUTINE TO DELETE L
INE
 560         JMP $A4A4
 570         ULERR JMP $A8E3
 580         END

END
```

100-110: la routine alla posizione A613 esadecimale converte il primo numero di linea prelevato da GETWRD in un indirizzo nella memoria di programma. Se il numero di linea non viene trovato, si azzererà il flag di riporto e verrà richiamato il messaggio di errore 'UNDEFINED LINE' (linea non definita).

130-160: l'indirizzo di linea scoperto dalla routine alla locazione A613 esadecimale è stato ora posto da quella routine nelle posizione 5F-60 esadecimali. Quei due byte vengono poi memorizzati sulla pila, dal momento che stiamo per ricavare un altro indirizzo di linea e non vogliamo perdere il primo.

170-190: si controlla che un '–' segua il primo numero di linea, usando la stessa routine che in qualche altro posto controlla una virgola, ma prima si definisce il carattere che stiamo cercando e si salta alla routine due byte più avanti che nel caso precedente.

200-230: si ricava l'ultimo numero di linea da cancellare e se ne determina l'indirizzo.

240-300: si ricava l'indirizzo della linea seguente l'ultima linea da cancellare mediante i byte di collegamento dell'ultima linea e lo si pone nei registri X ed Y.

310-410: l'indirizzo della linea seguente l'ultima viene ora piazzato nei byte di collegamento della prima linea da cancellare.

420-480: la serie di linee costituisce ora, agli occhi dell'interprete, un'unica linea, visto che i suoi byte di collegamento puntano oltre la fine della serie. Il numero reale della linea da cancellare si ricava ora dai due byte successivi a quelli di collegamento.

490-510: viene memorizzato zero nel buffer d'ingresso – questo indica al file editor del BASIC che si deve cancellare una linea.

520-540: una volta eliminate linee dal programma, non possiamo tornare allo stesso indirizzo da cui DELETE è stata chiamata, visto che ora potrebbe essere cambiato, quindi l'indirizzo di ritorno viene prelevato dalla pila ed eliminato.

550-560: si salta alla routine ROM che cancella le linee – essendo il corretto numero di linea memorizzato in 14 e 15 esadecimali per essere usato da questa routine.

570: salto alla routine di errore 'UNDEFINED LINE'

DELETE: listing completamente assemblato

```
add.  data    source code
0             10 PRT
0             20 ; BLOCK DELETE OF LINE
S
0             30 SYM
0             40 ORG $C053
C053  FFC3    50 WRD DEL-1
C055          60 ORG $C400
C400          70 GETWRD = $C12C
C400  202CC1  80 DEL JSR GETWRD
C403          90 ; CONVERT TO ADDRESS
C403  2013A6  100 JSR $A613
C406  903F    110 BCC ULERR
C408          120 ; SAVE POINTER ON STA
CK
C408  A55F    130 LDA $5F
C40A  48      140 PHA
C40B  A560    150 LDA $60
C40D  48      160 PHA
C40E          170 ; CHECK THAT A - SIGN
 FOLLOWS
C40E  A92D    180 LDA #45
C410  20FFAE  190 JSR $AEFF
C413          200 ; GET LAST NO. TO BE
DELETED
C413  202CC1  210 JSR GETWRD
C416  2013A6  220 JSR $A613
C419  902C    230 BCC ULERR
C41B          240 ; GET ADDRESS OF END
OF LAST LINE TO BE DELETED
C41B  A001    250 LDY #1
C41D  B15F    260 LDA ($5F).Y
C41F  AA      270 TAX
C420  88      280 DEY
C421  B15F    290 LDA ($5F).Y
C423  A8      300 TAY
C424          310 ; NOW STORE THESE BYT
ES IN FIRST LINE TO BE DELETED
C424  68      320 PLA
C425  8560    330 STA $60
C427  68      340 PLA
C428  855F    350 STA $5F
C42A  98      360 TYA
```

```
C42B  A000      370 LDY #0
C42D  915F      380 STA ($5F).Y
C42F  C8        390 INY
C430  8A        400 TXA
C431  915F      410 STA ($5F).Y
C433            420 ; GET LINE NO. TO BE
DELETED
C433  C8        430 INY
C434  B15F      440 LDA ($5F).Y
C436  8514      450 STA $14
C438  C8        460 INY
C439  B15F      470 LDA ($5F).Y
C43B  8515      480 STA $15
C43D            490 ; PUT ZERO INTO BASIC
 INPUT BUFFER - TELL FILE ED. TO DELETE
LINE
C43D  A900      500 LDA #0
C43F  8D0002    510 STA $200
C442            520 ; TIDY UP RETURN STAC
K
C442  68        530 PLA
C443  68        540 PLA
C444            550 ; USE ROM ROUTINE TO
DELETE LINE
C444  4CA4A4    560 JMP $A4A4
C447  4CE3A8    570 ULERR JMP $A8E3
C44A            580 END

 TOTAL ERRORS IN FILE --- 0


GETWRD           C12C
DEL              C400
ULERR            C447
 total number of symbols --- 3
```

DELETE: note sull'uso

Come ogni comando che altera la struttura di un programma, anche DELETE può creare problemi se usato durante l'esecuzione di un programma, così il programma termina quando esegue una DELETE. DELETE verrà usato normalmente in modo diretto, ma può essere usato come dispositivo di sicurezza nei programmi, ad esempio per rimuovere linee che non volete vengano esaminate, se viene pre-

muto il tasto STOP. Difatti, DELETE è più efficace nel proteggere un programma che ad esempio NEW ora che è disponibile UNDEAD. UNDEAD non può ripristinare le linee cancellate con DELETE, poiché sono già state riscritte.

La sintassi corretta per DELETE è:
DELETE <prima linea da cancellare> – <ultima linea da cancellare>

SEZIONE 5: BSAVE

Ora che, come speriamo, state appassionandovi alle meraviglie che è possibile realizzare con il codice macchina, desidererete presto poter salvare blocchi di memoria senza dover passare attraverso il monitor del programma Mastercode. In questa e nella prossima sezione presentiamo tre nuovi comandi che permetteranno di salvare, verificare e ricaricare in seguito una qualsiasi zona di memoria. Senza fare altre considerazioni, questo è un modo semplice per caricare routine macchina in memoria.

BSAVE: listing in linguaggio assemblatore

```
10        PRT
20        GETWRD = $C12C
30        SYM
40        ORG $C065
50        WRD BSAVE-1
60        ORG $C234
70        BSAVE JSR $E1D4
80        JSR $AEFD
90        JSR GETWRD
100        LDA $14
110        PHA
120        LDA $15
130        PHA
140        JSR $AEFD
150        JSR GETWRD
160        LDX $14
170        LDY $15
180        PLA
190        STA $15
```

```
 200          PLA
 210          STA $14
 220          LDA #$14
 230          JMP $E15F
 240          END

END
```

70: si usa una routine del KERNAL per prelevare tutti i parametri di un normale comando per file cioè nome, dispositivo ed indirizzo secondario.

100-140: l'inizio dell'area di memoria da salvare, prelevato da GETWRD, viene messo in pila.

150-210: l'indirizzo finale dell'area viene caricato nei registri X ed Y, l'indirizzo iniziale viene recuperato dalla pila e rimesso nelle posizioni 14-15 esadecimali.

220-230: viene caricato in accumulatore 14 esadecimale, per specificare l'indirizzo contenente l'indirizzo d'inizio del blocco di memoria da salvare. Il salvataggio, in sé è realizzato dalla stessa routine del KERNAL che esegue tutti i salvataggi da BASIC.

BSAVE: listing completamente assemblato

```
add.   data      source code
0                10 PRT
0                20 GETWRD = $C12C
0                30 SYM
0                40 ORG $C065
C065   33C2      50 WRD BSAVE-1
C067             60 ORG $C234
C234   20D4E1    70 BSAVE JSR $E1D4
C237   20FDAE    80 JSR $AEFD
C23A   202CC1    90 JSR GETWRD
C23D   A514      100 LDA $14
C23F   48        110 PHA
C240   A515      120 LDA $15
```

```
C242  48        130 PHA
C243  20FDAE    140 JSR $AEFD
C246  202CC1    150 JSR GETWRD
C249  A614      160 LDX $14
C24B  A415      170 LDY $15
C24D  68        180 PLA
C24E  8515      190 STA $15
C250  68        200 PLA
C251  8514      210 STA $14
C253  A914      220 LDA #$14
C255  4C5FE1    230 JMP $E15F
C258            240 END

 TOTAL ERRORS IN FILE --- 0


GETWRD            C12C
BSAVE             C234
 total number of symbols --- 2
```

BSAVE: note sull'uso

BSAVE richiede sempre un indirizzo secondario, che normalmente sarà 2. L'omissione dell'indirizzo secondario produrrà un SYNTAX ERROR. Notate anche che non c'è controllo sul fatto che l'indirizzo finale sia effettivamente seguente quello iniziale.

La sintassi corretta per BSAVE è:
BSAVE <"nome file">, <dispositivo>, <indirizzo secondario>, <inizio dell'area di memoria>, <fine dell'area di memoria>

SEZIONE 6: BLOAD e BVERIFY

Una volta salvata un'area di memoria su nastro o disco, sarebbe bello poterla recuperare. Questo è possibile con il comando BLOAD. Quasi la stessa routine che esegue BLOAD può venir usata per eseguire BVERIFY, che controlla se un'area

di memoria è stata salvata correttamente confrontando ciò che è stato salvato con l'area da cui è stato tratto.

BLOAD/BVERIFY: listing, in linguaggio assemblatore

```
10          PRT
20          SYM
25          GETWRD = $C12C
30          ORG $C061
40          WRD BLOAD-1.BVER-1
50          ORG $C2F2
60          BVER LDA #1
70          BYT $2C
80          BLOAD LDA #0
90          STA $A
100          JSR $E1D4
110          JSR $AEFD
120          JSR GETWRD
130          LDA $A
140          LDX $14
150          LDY $15
160          JMP $E175
170          END

END
```

100-160: è il nocciolo della routine e le due sezioni precedenti possono venir comprese solo dopo questa. Le linee saltano alla routine KERNAL che ricava i parametri per il file e chiamano GETWRD per ottenere l'indirizzo d'inizio dell'area di memoria in cui dev'essere caricato il codice salvayto. Il contenuto originale dell'accumulatore, che indica se dev'essere eseguito BLOAD o BVERIFY, viene ripristinato col risultato di GETWRD e viene chiamata la routine del KERNAL per il caricamento.

60-90: queste linee determinano il valore nell'accumulatore quando viene chiamata la routine di caricamento: 1 indicherà una verifica e 0 un caricamento. Se viene chiamato BLOAD, nell'accumulatore viene caricato uno zero. Se invece viene chiamato BVERIFY, si carica 1 nell'accumulatore e il byte seguente del programma (messo lì da una direttiva BYT assembler) viene interpretato come codice operativo di un'istruzione a tre byte con i due byte dell'istruzione LDA#0 come operando. Tutto ciò è ovviamente privo di senso, ma l'istruzione che viene riconosciuta

è un test su un bit che non provoca modifiche se non su un paio di flag della CPU che noi non usiamo. Ciò significa che l'istruzione in linea 60 viene superata molto più velocemente ed economicamente che se si fosse fatto un salto – l'istruzione scompare semplicemente quando è affrontata in questo modo.

BLOAD/BVERIFY: listing completamente assemblato

```
add.  data     source code
0              10 PRT
0              20 SYM
0              25 GETWRD = $C12C
0              30 ORG $C061
C061  F4C2F1   40 WRD BLOAD-1.BVER-1
C065           50 ORG $C2F2
C2F2  A901     60 BVER LDA #1
C2F4  2C       70 BYT $2C
C2F5  A900     80 BLOAD LDA #0
C2F7  850A     90 STA $A
C2F9  20D4E1   100 JSR $E1D4
C2FC  20FDAE   110 JSR $AEFD
C2FF  202CC1   120 JSR GETWRD
C302  A50A     130 LDA $A
C304  A614     140 LDX $14
C306  A415     150 LDY $15
C308  4C75E1   160 JMP $E175
C30B           170 END

 TOTAL ERRORS IN FILE --- 0


GETWRD          C12C
BVER            C2F2
BLOAD           C2F5
 total number of symbols --- 3
```

BLOAD/BVERIFY: note sull'uso

Ancora una volta si deve usare un indirizzo secondario, che dovrebbe essere zero. BLOAD ricaricherà a partire dall'indirizzo in memoria da voi specificato. BVERIFY verificherà l'area di memoria corretta, non importa quale indirizzo immettiate – comunque è necessario un indirizzo fittizio.

La sintassi corretta per BLOAD è:

BLOAD <“nome file”>, <dispositivo>, <indirizzo secondario, = 0>, <inizio dell’area in cui caricare>

La sintassi corretta per BVERIFY è:
BVERIFY <“nome file”>, <dispositivo>, <indirizzo secondario, = 0>, <valore fittizio>.

SEZIONE 7: MOVE

Quello che state per immettere, sebbene a prima vista possa sembrare un po’ scialbo, è uno dei comandi più flessibili che si possano aggiungere al BASIC. MOVE vi permette di specificare un’area di memoria e quindi spostarla indicando un nuovo punto d’inizio per quel blocco di memoria. Notate che le aree non vengono scambiate fra loro, al termine avrete due copie del blocco originale. Il comando può servire ai programmatori in linguaggio macchina che desiderano rilocare una routine senza doverla salvare e ricaricare. Può anche essere usata per manipolare lo schermo copiandone aree da un luogo ad un altro. La routine è più lunga della media fin qui incontrata, ma è veramente molto semplice come esecuzione.

MOVE: listing in linguaggio assemblatore

```
 10          PRT
 20          GETWRD = $C12C
 30          SYM
 40          ORG $C055
 50          WRD MOVE-1
 60          ORG $C30B
 70          ; BLOCK MOVE OF MEMORY - NO PR
OTETCION AGAINST MOVING VITAL SECTIONS
 80          ; SYNTAX OF COMMAND 'MOVE A1.A
2.L'
```

```
 90        ; WHERE A1 = ORIGINAL ADDRESS
 100        ;       A2 = FINAL ADDRESS
 110        ;       L  = LENGHT OF BLOCK
 120        ; ALSO NOTE 32K BLOCKS MAX
 130        NEWADD = $61
 140        OLDADD = NEWADD+2
 150        LENGHT = $14
 160        ; SUBROUTINE TO DECREMENT & T
EST LENGHT
 170        DECLEN LDA LENGHT
 180        BNE LBL000
 190        DEC LENGHT+1
 200        LBL000 DEC LENGHT
 210        LDA LENGHT
 220        ORA LENGHT+1
 230        RTS
 240        ; MAIN ROUTINE
 250        MOVE JSR GETWRD
 260        LDA $14
 270        PHA
 280        LDA $15
 290        PHA
 300        JSR $AEFD
 310        JSR GETWRD
 320        LDA $14
 330        PHA
 340        LDA $15
 350        PHA
 360        JSR $AEFD
 370        JSR GETWRD
 380        ; DECIDE WHICH DIRECTION TO M
OVE IN
 390        LDY #3
 400        LBL001 PLA
 410        STA NEWADD,Y
 420        DEY
 430        BPL LBL001
 440        LDA LENGHT
 450        ORA LENGHT+1
 460        BEQ LBL002
 470        LDA NEWADD+1
 480        CMP OLDADD+1
 490        BCC MVEDWN
 500        BNE MVEUP
 510        LDA NEWADD
 520        CMP OLDADD
 530        BCC MVEDWN
```

```
 540          ; MOVE BLOCK UPWARDS IN MEMOR
Y
 550          MVEUP CLD
 560          CLC
 570          LDA NEWADD
 580          ADC LENGHT
 590          STA NEWADD
 600          LDA NEWADD+1
 610          ADC LENGHT+1
 620          STA NEWADD+1
 630          CLC
 640          LDA OLDADD
 650          ADC LENGHT
 660          STA OLDADD
 670          LDA OLDADD+1
 680          ADC LENGHT+1
 690          STA OLDADD+1
 700          LDY #0
 710          LBL003 LDA (OLDADD).Y
 720          STA (NEWADD).Y
 730          TYA
 740          BNE LBL004
 750          DEC OLDADD+1
 760          DEC NEWADD+1
 770          LBL004 DEY
 780          JSR DECLEN
 790          BNE LBL003
 800          LBL002 RTS
 810          ; MOVE BLOCK DOWN THE MEMORY
 820          MVEDWN LDY #0
 830          LBL005 LDA (OLDADD).Y
 840          STA (NEWADD).Y
 850          INY
 860          BNE LBL006
 870          INC OLDADD+1
 880          INC NEWADD+1
 890          LBL006 JSR DECLEN
 900          BNE LBL005
 910          RTS

END
```

Commento

160-230: quando specificate un'area di memoria da spostare con MOVE, ovviamente essa avrà una certa lunghezza. Lo scopo di queste linee è di decrementare una variabile chiamata LENGTH per tener conto di quanta parte del blocco è stata finora trasferita. LENGTH è in realtà una variabile a due byte e le linee controllano se il byte basso è zero per decidere se decrementare solo il byte basso oppure entrambi – vengono decrementati entrambi se il byte basso vale zero per rappresentare un riporto.

240-370: i tre parametri per l'indirizzo d'inizio, fine e nuovo inizio vengono ricavati da GETWRD.

380-530: prima di eseguire una MOVE dobbiamo sapere se la destinazione è su o giù lungo la memoria. Se è 'giù' (cioè negativa in termini di indirizzo) dovremo iniziare a copiare l'area dal basso in modo che se le due aree si sovrappongono, al momento del contatto fra area di destinazione ed area sorgente non ci sia più bisogno dei dati all'inizio dell'area sorgente. Vale l'opposto quando l'area di destinazione è precedente in memoria rispetto all'area sorgente. Quindi se desiderassimo spostare un blocco di memoria di un byte più avanti inizieremmo a trasferire un byte in avanti l'ultimo byte dell'area sorgente. Cominciare all'inizio dell'area sorgente significherebbe che il primo byte verrebbe posto nella locazione del secondo, poi il secondo in quella del terzo e così via copiando lo stesso carattere nell'intero blocco di destinazione. Le linee fra 470 e 530 eseguono un confronto a 16 bit fra i due indirizzi iniziali e saltano a MVEDWN se l'indirizzo di destinazione è inferiore all'indirizzo iniziale sorgente, altrimenti si esegue MOVEUP.

550-690: è l'inizio della routine per spostare un blocco in memoria verso l'alto. Queste linee ricavano gli indirizzi finali di ciascun blocco e li memorizzano in NEWADD ed OLDADD, che hanno in precedenza contenuto gli indirizzi iniziali.

700-800: usando il registro Y per indicizzare MOVE, queste linee iniziano a spostare i byte dall'indirizzo specificato in OLDADD più il contenuto del registro Y all'indirizzo specificato da NEWADD più il registro Y. Il registro Y viene decrementato ad ogni trasferimento ed ogni volta che il contenuto di Y raggiunge zero i byte alti di OLDADD e NEWADD vengono decrementati di uno per accedere ad un nuovo blocco di 256 byte. Dopo ogni decremento del registro Y viene eseguito un salto alla subroutine DECLEN, che stabilisce se è stato trasferito l'intero blocco. Se sì, verrà posto a zero il flag di zero e si raggiungerà la linea 800, terminando la routine.

810-910: questa è la routine che sposta un blocco verso il basso. È più semplice poiché il contenuto di NEWDADD ed OLDADD può essere lasciato puntare all'inizio dei rispettivi blocchi. Oltre a questo, la sola vera differenza fra le due routine è che il registro Y viene incrementato e non decrementato.

MOVE: listing completamente assemblato

```
add.  data      source code
0               10 PRT
0               20 GETWRD = $C12C
0               30 SYM
0               40 ORG $C055
C055  17C3      50 WRD MOVE-1
C057            60 ORG $C30B
C30B            70 ; BLOCK MOVE OF MEMORY
 - NO PROTETCION AGAINST MOVING VITAL SE
CTIONS
C30B            80 ; SYNTAX OF COMMAND 'M
OVE A1.A2.L'
C30B            90 ; WHERE A1 = ORIGINAL
ADDRESS
C30B            100 ;      A2 = FINAL ADD
RESS
C30B            110 ;      L  = LENGHT OF
 BLOCK
C30B            120 ; ALSO NOTE 32K BLOCK
S MAX
C30B            130 NEWADD = $61
C30B            140 OLDADD = NEWADD+2
C30B            150 LENGHT = $14
C30B            160 ; SUBROUTINE TO DECRE
MENT & TEST LENGHT
C30B  A514      170 DECLEN LDA LENGHT
C30D  D002      180 BNE LBL000
C30F  C615      190 DEC LENGHT+1
C311  C614      200 LBL000 DEC LENGHT
C313  A514      210 LDA LENGHT
C315  0515      220 ORA LENGHT+1
C317  60        230 RTS
C318            240 ; MAIN ROUTINE
C318  202CC1    250 MOVE JSR GETWRD
C31B  A514      260 LDA $14
C31D  48        270 PHA
C31E  A515      280 LDA $15
C320  48        290 PHA
```

```
C321  20FDAE   300 JSR $AEFD
C324  202CC1   310 JSR GETWRD
C327  A514     320 LDA $14
C329  48       330 PHA
C32A  A515     340 LDA $15
C32C  48       350 PHA
C32D  20FDAE   360 JSR $AEFD
C330  202CC1   370 JSR GETWRD
C333           380 ; DECIDE WHICH DIRECT
ION TO MOVE IN
C333  A003     390 LDY #3
C335  68       400 LBL001 PLA
C336  996100   410 STA NEWADD.Y
C339  88       420 DEY
C33A  10F9     430 BPL LBL001
C33C  A514     440 LDA LENGHT
C33E  0515     450 ORA LENGHT+1
C340  F03C     460 BEQ LBL002
C342  A562     470 LDA NEWADD+1
C344  C564     480 CMP OLDADD+1
C346  9037     490 BCC MVEDWN
C348  D006     500 BNE MVEUP
C34A  A561     510 LDA NEWADD
C34C  C563     520 CMP OLDADD
C34E  902F     530 BCC MVEDWN
C350           540 ; MOVE BLOCK UPWARDS
IN MEMORY
C350  D8       550 MVEUP CLD
C351  18       560 CLC
C352  A561     570 LDA NEWADD
C354  6514     580 ADC LENGHT
C356  8561     590 STA NEWADD
C358  A562     600 LDA NEWADD+1
C35A  6515     610 ADC LENGHT+1
C35C  8562     620 STA NEWADD+1
C35E  18       630 CLC
C35F  A563     640 LDA OLDADD
C361  6514     650 ADC LENGHT
C363  8563     660 STA OLDADD
C365  A564     670 LDA OLDADD+1
C367  6515     680 ADC LENGHT+1
C369  8564     690 STA OLDADD+1
C36B  A000     700 LDY #0
C36D  B163     710 LBL003 LDA (OLDADD).Y
C36F  9161     720 STA (NEWADD).Y
C371  98       730 TYA
C372  D004     740 BNE LBL004
```

```
C374  C664      750 DEC OLDADD+1
C376  C662      760 DEC NEWADD+1
C378  88        770 LBL004 DEY
C379  200BC3    780 JSR DECLEN
C37C  D0EF      790 BNE LBL003
C37E  60        800 LBL002 RTS
C37F            810 ; MOVE BLOCK DOWN THE
 MEMORY
C37F  A000      820 MVEDWN LDY #0
C381  B163      830 LBL005 LDA (OLDADD).Y
C383  9161      840 STA (NEWADD).Y
C385  C8        850 INY
C386  D004      860 BNE LBL006
C388  E664      870 INC OLDADD+1
C38A  E662      880 INC NEWADD+1
C38C  200BC3    890 LBL006 JSR DECLEN
C38F  D0F0      900 BNE LBL005
C391  60        910 RTS

 TOTAL ERRORS IN FILE --- 0


GETWRD          C12C
NEWADD          61
OLDADD          63
LENGHT          14
DECLEN          C30B
LBL000          C311
MOVE            C318
LBL001          C335
MVEUP           C350
LBL003          C36D
LBL004          C378
LBL002          C37E
MVEDWN          C37F
LBL005          C381
LBL006          C38C
 total number of symbols --- 15
```

MOVE: note sull'uso

L'uso di MOVE è abbastanza evidente, ma ricordate che non c'è protezione se fate qualche stupidaggine, tipo scrivere accidentalmente sull'interprete (quando è in

RAM) o su variabili di sistema o nell'area di programma o... Assicuratevi di sapere cosa state trasferendo e cosa c'è nel punto di destinazione PRIMA di fare il movimento.

La sintassi corretta per MOVE è:
MOVE <indirizzo destinazione>, <indirizzo di partenza>, <lunghezza>

SEZIONE 8: FILL

Dopo aver progettato MOVE, il logico sviluppo era FILL, che viene usata per riempire una specifica area di memoria con un certo valore. Può venir usata per pulire aree di memoria, aree dello schermo, per cambiare le caratteristiche cromatiche dello schermo riempiendo parti del file attributi. Il vero lavoro è compiuto da una chiamata alla routine MVEDWN in MOVE, quindi non c'è molto da spiegare sul funzionamento del comando in sè.

FILL: listing in linguaggio assemblatore

```
10        PRT
20        SYM
30        ORG $C067
40        WRD FILL-1
50        GETWRD = $C12C
60        MVEDWN = $C37F
70        DECLEN = $C30B
80        ORG $C392
90        FILL JSR GETWRD
100        LDA $14
110        PHA
120        LDA $15
130        PHA
140        JSR $AEFD
```

```
150         JSR GETWRD
160         LDA $14
170         PHA
180         LDA $15
190         PHA
200         JSR $AEFD
210         JSR GETWRD
220         LDA $15
230         BNE IQERR
240         LDX $14
250         PLA
260         STA $15
270         PLA
280         STA $14
290         PLA
300         STA $62
310         STA $64
320         PLA
330         STA $61
340         STA $63
350         INC $61
360         BNE LBL000
370         INC $62
380         LBL000 LDY #0
390         TXA
400         STA ($63).Y
410         JSR DECLEN
420         BEQ EXIT
430         JMP MVEDWN
440         IQERR JMP $B248
450         EXIT RTS
460         END

END
```

Commento

100-230: queste linee ottengono i tre paramentri dell'indirizzo iniziale, della lunghezza e del valore da porre in ciascun byte del blocco. Viene generato un errore del tipo 'ILLEGAL QUANTITY' se il valore da caricare è maggiore di 255.

240: il valore da salvare nel blocco viene posto nel registro X.

250-280: la lunghezza del blocco è memorizzata nelle locazioni 14-15 esadecimali, dove la routine move si aspetta di trovarli.

290-370: l'indirizzo d'inizio viene posto in OLDADD e l'indirizzo iniziale più uno in NEWADD.

380-400: il valore da usare nel riempire il blocco è posto nel primo byte del blocco.

410-420: la lunghezza è decrementata di uno e se la lunghezza era solo uno la routine termina:

430: chiama MVEDWN. Se ricordate qualcosa del commento su MOVE saprete che spostare un byte verso l'alto in memoria implica una chiamata a MOVEUP. Chiamare MOVDWN allo scopo ha la tremenda conseguenza di rovinare proprio i byte da trasferire, riscrivendo costantemente il byte uno nel byte due, il due nel tre e così via riempiendo l'intera area con lo stesso valore.
In MOVE sarebbe stato un disastro, mentre è proprio ciò che vogliamo in FILL.

FILL: listing completamente assemblato

```
add.   data     source code
0               10 PRT
0               20 SYM
0               30 ORG $C067
C067   91C3     40 WRD FILL-1
C069            50 GETWRD = $C12C
C069            60 MVEDWN = $C37F
C069            70 DECLEN = $C30B
C069            80 ORG $C392
C392   202CC1   90 FILL JSR GETWRD
C395   A514     100 LDA $14
C397   48       110 PHA
C398   A515     120 LDA $15
C39A   48       130 PHA
C39B   20FDAE   140 JSR $AEFD
C39E   202CC1   150 JSR GETWRD
C3A1   A514     160 LDA $14
C3A3   48       170 PHA
C3A4   A515     180 LDA $15
C3A6   48       190 PHA
```

```
C3A7  20FDAE   200 JSR $AEFD
C3AA  202CC1   210 JSR GETWRD
C3AD  A515     220 LDA $15
C3AF  D025     230 BNE IQERR
C3B1  A614     240 LDX $14
C3B3  68       250 PLA
C3B4  8515     260 STA $15
C3B6  68       270 PLA
C3B7  8514     280 STA $14
C3B9  68       290 PLA
C3BA  8562     300 STA $62
C3BC  8564     310 STA $64
C3BE  68       320 PLA
C3BF  8561     330 STA $61
C3C1  8563     340 STA $63
C3C3  E661     350 INC $61
C3C5  D002     360 BNE LBL000
C3C7  E662     370 INC $62
C3C9  A000     380 LBL000 LDY #0
C3CB  8A       390 TXA
C3CC  9163     400 STA ($63).Y
C3CE  200BC3   410 JSR DECLEN
C3D1  F006     420 BEQ EXIT
C3D3  4C7FC3   430 JMP MVEDWN
C3D6  4C48B2   440 IQERR JMP $B248
C3D9  60       450 EXIT RTS
C3DA           460 END

 TOTAL ERRORS IN FILE --- 0


GETWRD          C12C
MVEDWN          C37F
DECLEN          C30B
FILL            C392
LBL000          C3C9
IQERR           C3D6
EXIT            C3D9
 total number of symbols --- 7
```

FILL: note sull'uso

Ancora una volta, il comando non protegge dalle stupidaggini.

La sintassi corretta per FILL è:

FILL <indirizzo iniziale>, <lunghezza>, <valore del byte>

SEZIONE 9: RESTORE

Potreste pensare, vedendo il titolo della sezione, che siamo usciti di senno. Non c'è già un comando RESTORE nel BASIC normale? La risposta è ovviamente si, ma una delle possibilità affascinanti aperte dallo spostamento dell'interprete in RAM è che non solo possiamo aggiungere nuovi comandi, ma anche cambiare quelli esistenti.

RESTORE è un candidato principe a questo cambiamento. La normale routine di RESTORE risale al tempo in cui i computer erano conservati in grandi sale con aria condizionata e leggevano i loro programmi e dati da schede perforate. Ora il fatto è che con una pila di schede potete solo leggere dall'inizio. Se volete trovare la scheda n. 97, dovete iniziare dalla uno e leggervi tutte le 96 schede che non vi interessano. Non c'è alcun motivo per cui debba essere così anche su un moderno microcomputer, tuttavia pare che si sia radicata la convinzione che l'unico modo di trattare le frasi DATA sia quello di cominciare all'inizio e proseguire fino al punto voluto.

In questa sezione modificheremo il normale comando RESTORE in modo che possiate ripristinare un numero di linea specificato e prelevare il primo dato che segue DATA (il normale comando RESTORE può venir usato ancora se necessario). In questo modo potete ordinare i vostri dati in tabelle separate e saltare esattamente alla tabella che volete, risparmiando considerevolmente tempo per accedere a singoli elementi di frasi DATA rendendo inoltre il funzionamento del vostro programma più trasparente.

RESTORE: listing in linguaggio assemblatore

```
 10         PRT
 20         SYM
 30         ORG $C132
 40         ; ALTER TO RESTORE TO LINE NOS
.
 50         ; TO USE TRANSFER ROM TO RAM A
ND ALTER RESTORE VECTOR TO 'START-1'
 60         ; RESTORE VECTOR AT $A022
 70         GETWRD = $C12C
 80         START LDA #0
 90         STA $14
```

```
100         STA $15
110         JSR $73
120         LDA $7A
130         BNE LBL000
140         DEC $7B
150         LBL000 DEC $7A
160         BCS LBL001
170         JSR GETWRD
180         LBL001 JSR $A613
190         LDA $14
200         ORA $15
210         BEQ LBL002
220         BCC ULERR
230         LBL002 LDA $5F
240         LDY $60
250         SEC
260         SBC #1
270         JMP $A824
280         ULERR JMP $A8E3
290         END

END
```

Commento

80-100: inizializzazione della locazione in cui l'interprete memorizzerà poi un numero di linea.

110-150: la chiamata a subroutine preleva il carattere successivo del testo BASIC e le altre linee supportano il puntatore di testo che è stato ora posto nella posizione seguente a quel carattere.

160: se il carattere prelevato dalla routine alla locazione 73 esadecimale è una cifra, verrà posto uguale a uno il flag di riporto. Se non lo è, viene fatto un salto oltre GETWRD, poiché la routine supporrà che debba venir eseguito un normale RESTORE e non si debba reperire alcun numero di linea. Se desiderate usare un'espressione dopo RESTORE dovrete precederla con 00+ o con 01* in modo da assicurare la chiamata a GETWRD.

180: questa routine trova l'indirizzo del numero di linea ottenuto da GETWRD o, se GETWRD non è stata chiamata, della prima linea nel programma BASIC.

190-210: se l'area di memorizzazione del numero di linea contiene zero, si suppone che sia stata eseguita una RESTORE normale e non si fanno controlli sui parametri.

220: se, al ritorno da A613, il flag di riporto è a zero, il numero di linea referenziato non è stato trovato e viene segnalato un errore di tipo 'UNDEFINED LINE'

230-270: l'indirizzo della linea trovata viene prelevato dalle locazioni esadecimali 5F-60, da esso viene sottratto uno e l'esecuzione torna alla normale routine di RESTORE, col puntatore ai dati però rivolto ora alla linea specificata.

RESTORE: listing completamente assemblato

```
add.  data      source code
0               10 PRT
0               20 SYM
0               30 ORG $C132
C132            40 ; ALTER TO RESTORE TO
LINE NOS.
C132            50 ; TO USE TRANSFER ROM
TO RAM AND ALTER RESTORE VECTOR TO 'STAR
T-1'
C132            60 ; RESTORE VECTOR AT $A
022
C132            70 GETWRD = $C12C
C132  A900      80 START LDA #0
C134  8514      90 STA $14
C136  8515      100 STA $15
C138  207300    110 JSR $73
C13B  A57A      120 LDA $7A
C13D  D002      130 BNE LBL000
C13F  C67B      140 DEC $7B
C141  C67A      150 LBL000 DEC $7A
C143  B003      160 BCS LBL001
C145  202CC1    170 JSR GETWRD
C148  2013A6    180 LBL001 JSR $A613
```

```
C14B  A514      190 LDA $14
C14D  0515      200 ORA $15
C14F  F002      210 BEQ LBL002
C151  900A      220 BCC ULERR
C153  A55F      230 LBL002 LDA $5F
C155  A460      240 LDY $60
C157  38        250 SEC
C158  E901      260 SBC #1
C15A  4C24A8    270 JMP $A824
C15D  4CE3A8    280 ULERR JMP $A8E3
C160            290 END

 TOTAL ERRORS IN FILE --- 0


GETWRD           C12C
START            C132
LBL000           C141
LBL001           C148
LBL002           C153
ULERR            C15D
 total number of symbols --- 6
```

RESTORE: note sull'uso

RESTORE può venir usata normalmente se non si specifica un parametro o si pone zero come parametro. Una limitazione è che la RESTORE estesa lavorerà solo su parametri di più di una cifra. Se desiderate ripristinare una linea con numero inferiore a 10, dovrete aggiungere uno zero iniziale al numero.

La sintassi corretta per RESTORE è, nella forma estesa:
RESTORE numero di linea di almeno due cifre.

Prima di usare la RESTORE estesa si deve eseguire un'ulteriore modifica al programma BASIC Extender II. Si devono aggiungere le seguenti linee:

```
250 REM ALTER RESTORE VECTOR
251 POKE 40996,49: POKE 40997,193
```

La routine in linguaggio macchina dovrebbe essere assemblata nel file macchina dell'estender e delle nuove routine e quindi caricata in memoria usando il programma BASIC extender II appena modificato.

CAPITOLO 10
FUNZIONI BASIC

Le funzioni BASIC, come ricorderete, ricevono da parte dell'interprete un trattamento separato, essendo identificate durante l'esame di un programma, dal fatto che esse occupano un ben specificato segmento della tavola delle parole chiave. La principale differenza fra le parole chiave per azioni e quelle per funzioni è che le funzioni richiederanno la valutazione di qualcosa, per cui c'è un'intera serie di routine interprete speciali. Di fatto circa la metà dell'interprete è dedicato al problema di valutare espressioni in un modo o nell'altro. L'esecuzione di una parola funzione non è semplicemente una faccenda di salti a singole routine macchina in memoria.

La routine macchina data qui, sebbene corta, usa routine dell'interprete che sono di gran lunga più complesse che quelle usate dalle parole azioni. Se queste routine non fossero già presenti nell'interprete sarebbe stata un'impresa enorme definire nuove funzioni.

Valutatore di espressioni esteso: listing in linguaggio assemblatore

```
 10        PRT
 20        SYM
 30        ORG $C44A
 40        ; EXTEND EXPRESSION EVALUATOR
 50        ; TO USE POKE $AFAA WITH 'JMP
FUNEVL'
 60        FUNEVL CPX #$8F
 70        BCC LBL002
 80        CPX #$98
 90        BCC LBL001
 100        CPX #$9F
```

```
110          BCS LBL001
120          JSR $AEF1
130          PLA
140          TAX
150          CPX #$98
160          BEQ DEEK
170          CPX #$9A
180          BEQ YPOS
190          BNE VARPTR
200          LBL001 JMP $AFB1
210          LBL002 JMP $AFD1
220          VARPTR LDA 72
230          LDY 71
240          LBL003 JSR $B391
250          LDA $66
260          BPL LBL004
270          LDY #CONST/256
280          LDA #CONST-CONST/256*256
290          JSR $BA8C
300          JSR $B86A
310          LBL004 JMP $AD8D
320          YPOS SEC
330          JSR $FFF0
340          TXA
350          TAY
360          LDA #$0
370          JMP LBL003
380          ; PERFORM DEEK
390          DEEK JSR $B7F7
400          LDY #1
410          LDA ($14).Y
420          PHA
430          DEY
440          LDA ($14).Y
450          TAY
460          PLA
470          JMP LBL003
480          CONST BYT 145.0.0.0.0
490          END

END
```

60-70: controlla se il token è quello di una normale funzione numerica. In questo caso si chiama il valutatore normale.

80-110: si controlla se la funzione il cui token è nel registro X è nell'intervallo delle nostre tre nuove funzioni, altrimenti si chiama il valutatore di stringhe funzioni.

120: questa routine dell'interprete valuta un'espressione fra parentesi, cioè l'argomento della funzione.

130-140: il valutatore di espressioni chiamato nella linea precedente pone il token sulla cima della pila. Ora viene recuperato e memorizzato nel registro X.

150-190: a questo punto il token dev'essere quello di una delle nostre nuove funzioni, queste linee determinano quale e saltano alla routine appropriata.

220-310: VARPTR: questa funzione restituisce un puntatore alla locazione in memoria di ogni variabile il cui nome sia posto fra parentesi come argomento.

220-230: al ritorno dalla routine situata alla locazione AEF1, che valuta la variabile fra parentesi, l'indirizzo della variabile è contenuto nelle posizioni 47-48 esadecimale (71-72 decimale). Il valore a due byte viene caricato in accumulatore e nel registro Y.

240: questo indirizzo viene ora inviato alla routine che converte questo numero intero in numero a virgola mobile. Questo è necessario poiché l'interprete si aspetta un numero in virgola mobile come risultato di una funzione, e lo tratterà di conseguenza.

250-300: sfortunatamente, il convertitore in virgola mobile cambierà un numero intero senza segno in uno a virgola mobile con segno. Ciò significa che i numeri superiori a 32767 risulteranno negativi, come ad es. avviene per la funzione FRE nel BASIC normale. Se immettete FRE (0) quando non c'è un programma in memoria, verrà restituita una quantità negativa, a cui va aggiunto 65536 per avere il risultato corretto. Queste linee controllano se c'è un segno meno memorizzato nel byte di segno dell'accumulatore #1 della virgola mobile (alla locazione 66 esadecimale, in pagina zero). Se si trova un meno (cioè se il bit 7 vale uno), si carica in accumulatore e nel registro Y l'indirizzo della variabile CONSTANT (linea 480), che in realtà è 65536 in virgola mobile. Questo valore viene ora posto nell'accumulatore #2 della virgola mobile con una chiamata alla routine dell'interprete situata in BA8C (esadecimale) e sommato al valore in accumulatore #1 della virgola mobile da una chiamata alla routine posta in B86A.

310: si torna al valutatore di espressioni, che potrebbe essere ancora nella valutazione di un'espressione di cui VARPTR è solo una parte.

320-370: YPOS: questa nuova funzione restituisce la posizione verticale del cursore sullo schermo. È parallela alla normale funzione BASIC POS, che restituisce la posizione orizzontale del cursore.

330: la routine del KERNAL che restituisce la posizione del cursore nei registri X ed Y.

340-350: il contenuto del registro X viene memorizzato nel registro Y (il registro X conteneva originariamente la posizione verticale).

360-370: l'accumulatore ed il registro Y contengono ora lo stesso valore. Si carica zero in accumulatore, fornendo così, nell'accumulatore e nel registro Y, un numero a 16 bit nell'intervallo 0-24 (i numeri delle linee sullo schermo). Questo intero a 16 bit viene ora inviato alla routine situata a LBL003, che lo converte in numero a virgola mobile.

390-470: DEEK: questa funzione restituisce un numero nell'intervallo 0-65535. È di fatto una PEEK doppia ed equivale alla frase in BASIC normale: PEEK (X) +256*PEEK (X+1).

390: il parametro valutato dal valutatore di funzioni viene convertito in un intero con questa chiamata (PEEK e DEEK si rivolgono ad indirizzi interi).

400-460: il valore restituito si dovrà trovare nei nostri vecchi amici 14-15 esadecimali. L'accumulatore ed il registro Y vengono caricati con questo valore e si salta a LBL003, restituendo così il contenuto dei due byte al valutatore di espressioni.

Valutatore di espressioni esteso: listing completamente assemblato

```
add.    data        source code
0                   10 PRT
0                   20 SYM
0                   30 ORG $C44A
```

```
C44A              40 ; EXTEND EXPRESSION EV
ALUATOR
C44A              50 ; TO USE POKE $AFAA WI
TH 'JMP FUNEVL'
C44A  E08F        60 FUNEVL CPX #$8F
C44C  901A        70 BCC LBL002
C44E  E098        80 CPX #$98
C450  9013        90 BCC LBL001
C452  E09F        100 CPX #$9F
C454  B00F        110 BCS LBL001
C456  20F1AE      120 JSR $AEF1
C459  68          130 PLA
C45A  AA          140 TAX
C45B  E098        150 CPX #$98
C45D  F02F        160 BEQ DEEK
C45F  E09A        170 CPX #$9A
C461  F020        180 BEQ YPOS
C463  D006        190 BNE VARPTR
C465  4CB1AF      200 LBL001 JMP $AFB1
C468  4CD1AF      210 LBL002 JMP $AFD1
C46B  A548        220 VARPTR LDA 72
C46D  A447        230 LDY 71
C46F  2091B3      240 LBL003 JSR $B391
C472  A566        250 LDA $66
C474  100A        260 BPL LBL004
C476  A0C4        270 LDY #CONST/256
C478  A99E        280 LDA #CONST-CONST/256*
256
C47A  208CBA      290 JSR $BA8C
C47D  206AB8      300 JSR $B86A
C480  4C8DAD      310 LBL004 JMP $AD8D
C483  38          320 YPOS SEC
C484  20F0FF      330 JSR $FFF0
C487  8A          340 TXA
C488  A8          350 TAY
C489  A900        360 LDA #$0
C48B  4C6FC4      370 JMP LBL003
C48E              380 ; PERFORM DEEK
C48E  20F7B7      390 DEEK JSR $B7F7
C491  A001        400 LDY #1
C493  B114        410 LDA ($14).Y
C495  48          420 PHA
C496  88          430 DEY
C497  B114        440 LDA ($14).Y
C499  A8          450 TAY
C49A  68          460 PLA
C49B  4C6FC4      470 JMP LBL003
```

```
C49E  910000    480 CONST BYT 145.0.0.0.0
C4A3            490 END

 TOTAL ERRORS IN FILE --- 0


FUNEVL          C44A
LBL001          C465
LBL002          C468
VARPTR          C46B
LBL003          C46F
LBL004          C480
YPOS            C483
DEEK            C48E
CONST           C49E
 total number of symbols --- 9
```

VARPTR: note sull'uso

Se si mette fra parentesi un'espressione invece di una variabile, il risultato è privo di significato e non dovrebbe venir usato per cambiamenti in memoria. Oltre a ciò la funzione può essere usata come scorciatoia per cambiare singoli caratteri in arrays di stringhe (evitando problemi di garbage collection e complesse funzioni stringa) o semplicemente per avere un'idea più precisa di cosa accade nell'area delle variabili. Come esempio dell'uso di VARPTR, immettete A%=10 in modo diretto, poi PRINT DEEK (VARPTR(A%)+1). Questo restituirà il valore di A%.

Nel caso di stringhe, PEEK (VARPTR(A$)) restituirà la lunghezza di A$.

DEEK(VARPTR(A$)+1) restituisce l'indirizzo inziale di A$ in memoria.

La sintassi corretta per VARPTR è:
VARPTR(<nome della variabile>)

YPOS: note sull'uso

Questa funzione è parallela a POS nel BASIC normale, compreso il fatto che la variabile fra parentesi è ignorata.

La sintassi corretta per YPOS è:
YPOS(<argomento fittizio>)

DEEK: note sull'uso

Molto simile a PEEK, eccetto che restituisce un valore a due byte. DEEK è particolarmente utile per accedere ai valori memorizzati in registri a due byte usati dal

sistema, es. DEEK(43) restituisce l'indirizzo di inizio BASIC.
La sintassi corretta per DEEK è:
DEEK(<espressione>)

Uso dell'estensore di funzioni

Come per le parole chiave del BASIC esteso, questa routine deve essere assemblata sul file macchina che avete creato per includere l'estensore del BASIC e gli altri comandi. La routine potrebbe stare da sola senza interrompere il funzionamento del sistema se fosse caricata in memoria, ma non sareste in grado di creare i token per le nuove funzioni.

Allo scopo di inserire la routine nel valutatore di funzioni esistente, la linea seguente va aggiunta al vostro programma BASIC Extender II:

```
224 DATA 173,175,76,174,175,74,175,175,196
```

L'effetto di questa linea è di piazzare nel valutatore di funzioni un salto alla nostra routine macchina in modo simile a quanto fatto per la routine di esecuzione di parole azioni.

Una volta fatta la modifica, il valutatore di funzioni può venir caricato, con gli altri comandi, dal programma BASIC Extender II.

CAPITOLO 11
NUOVE FRONTIERE

Quando iniziate a scrivere un libro di questo genere, buttate lì tutte le idee per routine che vorreste includere. Alcune si dimostrano eccessivamente estese quanto alla codifica, altre sembrano di scarso rilievo ad un'attenta considerazione. Alcune però si impongono come idee utili che non dovrebbero presentare troppe difficoltà nella loro realizzazione. Un'idea di questo genere che ci è venuta — lo ammettiamo: a notte fonda — è FAST, una routine che fa tesoro di RKILL rimuovendo i controlli che l'interprete fa per cercare gli spazi durante l'esecuzione di un programma. Il 64 fu acceso, la routine immessa ed assemblata, aggiungendo due nuove parole FAST e SLOW (la seconda si limita a ripristinare lo stato normale delle cose). La routine venne in questo modo:

FAST e SLOW: listing in linguaggio assemblatore

```
10        PRT
20        SYM
30        ORG $C057
40        WRD FAST-1.SLOW-1
45        ORG $C4A3
50        SLOW LDY #0
60        BYT $2C
70        FAST LDY #4
80        LDX #0
90        LBL000 LDA $E3AF.Y
 100        STA $80.X
 110        INX
 120        INY
 130        CPY #$B
 140        BCC LBL000
 150        RTS
 160        END

END
```

Fu caricato il BASIC Extender e FAST venne messa in memoria.

La memoria venne pulita e fu immesso un piccolo programma che eseguiva un grosso ciclo, impiegando in totale circa 57,75 secondi. Fu eseguita RKILL sul programma, rimuovendo tutti gli spazi poi fu dato FAST in modo diretto. Dal 64 nemmeno un sospiro, la routine stava ovviamente per funzionare fin dalla prima volta — un trionfo della pianificazione e progettazione accurata dei programmi. Il ciclo venne fatto eseguire ancora, con un cronometro nelle mani tremanti.

Il risultato fu un tempo di 57,25 secondi, un miglioramento dello 0,89%!

Non tutto ciò che si può fare in codice macchina vale la pena farlo, né tutto ciò che vale la pena fare è possibile. Al termine di questo libro ci è rimasta la sensazione che ciò che abbiamo creato valesse la pena crearlo e che altre persone saranno in grado di farlo e di imparare da esso. Introducendo il programma Mastercode e le routine per le parole chiave (e comprendendoli) avrete compiuto, speriamo, un considerevole passo avanti nella comprensione del vostro 64 e del potenziale che offre per la programmazione in linguaggio macchina. Possano i vostri sforzi avere maggior successo di FAST.

APPENDICE A
GENERATORE DELLE SOMME DI CONTROLLO

Il programma che segue è quello usato per generare le tavole di controllo fornite insieme a ciascun modulo del programma Mastercode. La cifra di controllo per ogni linea è un metodo quasi a prova d'errore di indicazione della correttezza di una linea. Il programma lavora sommando i valori di tutti i byte della linea, tornando a zero ogni volta che si raggiunge 255. Un carattere scorretto o omesso provocherà un cambiamento nella somma. Per far uso delle somme di controllo, immettete questo programma nel 64 *prima* di iniziare il programma Mastercode e poi date RUN 63800 ogni volta che vorrete controllare un gruppo di linee che avete immesso, quindi confrontate il risultato con la tavola di controllo del modulo. Se c'è qualche differenza, la linea che avete immesso è diversa da quella del libro. Notate che gli spazi contano nel calcolo delle somme.

```
63800 REM CHECKSUM PROGRAM
63801 GOSUB 63810
63802 GOSUB 63840
63803 IF FL>=0 THEN 63802
63804 END
63810 DEFFN DEEK(X) = PEEK(X)+256*PEEK(X
+1)
63820 REM DATA FOR MACHINE CODE
63821 DATA ***
63822 DATA 165,252,166,253,133,020,134,0
21,032,019
63823 DATA 166,216,160,001,177,095,133,2
54,240,013
63824 DATA 200,177,095,133,252,200,177,0
95,133,253
63825 DATA 200,169,000,133,251,177,095,2
40,006,024
63826 DATA 101,251,200,208,244,096
63827 DATA -1
```

```
63830 REM PUT DATA INTO MEMORY
63831 AD = 52992
63832 RESTORE
63833 READ T$: IF T$<>"***" THEN 63833
63834 READ T : IF T>=0 THEN POKE AD,T :
AD = AD+1 : GOTO 63834
63835 DEV = 3 : IN$ = "" : INPUT "OUTPUT
 DEVICE NUMBER "; DEV
63836 IF DEV=1 OR DEV>4 THEN INPUT "FILE
 NAME ";IN$
63837 R$ = CHR$ (13) : S$ = "***********
****************************"+R$
63838 RETURN
63840 REM DO INITIALISATION
63841 FL = 0 : INPUT "FIRST LINE "; FL :
 IF FL<0 THEN RETURN
63842 LL = 65536 : INPUT "LAST LINE "; L
L
63843 INPUT "MODULE NAME ";M$
63844 IF DEV=1 OR DEV>4 THEN OPEN 1,DEV,
2,IN$ : GOTO 63846
63845 OPEN 1,DEV
63846 PRINT#1,S$ R$SPC((40-LEN(M$))/2)M$
 R$ R$"LINE NUMBERS"FL"TO"LL;R$S$R$
 63850 REM ACTUAL PROGRAM
 63851 LN = FL : C = 0 : C1 = 0
 63852 POKE 252,LN-INT(LN/256)*256 : POKE
  253,LN/256
 63853 SYS 52992 : CS = PEEK(251) : LN =
 FNDEEK(252)+1
 63860 REM FORMAT OUTPUT INTO 3 COLUMNS
 63861 T$ = LEFT$ (STR$(LN-1)+"      ",6)
 +LEFT$ (STR$(CS)+"       ",7)
 63862 PRINT#1,T$;
 63864 C = C+1 : IF C>=3 THEN PRINT#1 : C
  = 0 : C1 = C1 + 1
 63865 IF C1>=20 AND DEV=3 THEN C1 = 0 :
 GOSUB 63998
 63866 IF LN<=LL AND PEEK(254) THEN 63852
 63867 PRINT#1,R$ : CLOSE 1 : RETURN
 63898 GET T$ : IF T$="" THEN 63998
 63899 RETURN
```

```
63800 58        63801 175       63802 178
63803 186       63804 128       63810 179
63820 32        63821 33        63822 30
63823 38        63824 46        63825 31
63826 14        63827 1         63830 53
63831 130       63832 140       63833 176
63834 38        63835 214       63836 104
63837 145       63838 142       63840 131
63841 3         63842 133       63843 188
63844 118       63845 251       63846 105
63850 161       63851 13        63852 207
63853 255       63860 119       63861 189
63862 168       63864 79        63865 78
63866 206       63867 222       63898 198
63899 142
```

APPENDICE B
MASTERCODE: GUIDA DELL'UTENTE

Il programma Mastercode si divide in quattro sezioni: Monitor, Disassembler, File Editor ed Assembler, tutte perfettamente compatibili fra loro. Al momento dell'inizio dell'esecuzione di Mastercode, ci sarà una notevole attesa mentre vengono generate le complesse tabelle per il Disassembler e l'Assembler. La prima sezione del programma disponibile per l'utente è il Monitor - Assembler, Disassembler e File Editor sono tutti richiamati dal Monitor come opzioni del menu.

Monitor

L'uso del Monitor è semplicissimo. Basta seguire le indicazioni date quando il programma viene fatto eseguire. Disassembler e Assembler tornano entrambi al Monitor una volta terminati i loro compiti correnti. Il Monitor è l'opzione 0 durante l'uso del File Editor.

Disassembler

Il Disassembler è in grado di fornire traduzioni in linguaggio assemblatore di tutte le istruzioni macchina del 6502/6510 nel formato standard stabilito dalla Mostechnology (ora parte della Commodore Semiconductor Group), progettista del chip.

Per usare il Disassembler è necessario solo specificare, in esadecimale, l'indirizzo d'inizio dell'area di memoria da disassemblare. È necessaria una certa attenzione nella scelta del punto d'inizio corretto, dal momento che un indirizzo iniziale non coincidente con il primo byte di un'istruzione macchina risulterà in uno o più '???' o istruzioni tradotte erroneamente prima che il disassembler si sincronizzi con la memoria. La ricorrenza di '???' nel mezzo di un'area disassemblata di memoria indica la presenza di tabelle di dati. Le istruzioni disassemblate circondate da indicatori '???' vanno prese con cautela, poiché possono rappresentare byte casuali che somigliano a vere istruzioni macchina. Al termine di tabelle di dati si possono

verificare altri disguidi, per la stessa ragione del caso in cui si sia specificato un punto di partenza non valido. Quando si disassemblano zone successive a tabelle, è bene tentare di identificare la fine della tabella iniziando con l'indirizzo dell'ultimo '???' e facendo diversi disassemblaggi, oguno che inizi dal byte successivo, fino a trovare un punto di partenza che generi linguaggio assemblatore sensato fin dal principio.

File Editor

Il file editor è semplicemente un mezzo di immettere una serie di linee numerate — non si fanno controlli se si immettano istruzioni valide del linguaggio assemblatore. Si possono inserire linee all'interno di file esistenti dando loro numeri appropriati. Si possono listare e cancellare blocchi di linee.

Si possono cancellare linee singole durante l'immissione battendo il numero di linea senza testo seguente (come in BASIC).

Si possono salvare file non assemblati per un successivo caricamento. Si possono fondere file da memoria esterna con file già presenti in memoria centrale, se la lunghezza totale risultante non è superiore a 255 linee — ogni linea può contenere solo una istruzione in linguaggio assemblatore. Si possono numerare le linee di un file da fondere con un altro già in memoria in modo che cadano all'interno del testo presente, all'inizio o alla fine o anche rimpiazzino linee già esistenti. La possibilità di 'change device' (cambio di dispositivo) permette di alterare il numero del dispositivo corrente di I/O, permettendo così di salvare file sul dispositivo 4 (stampante) o di fare copie di sicurezza su nastro per coloro che lavorano su disco. Non c'è controllo che il dispositivo di I/O corrente sia connesso o sia in grado di salvare o caricare. Possono essere aggiunti a un file dati in codice macchina dalla memoria, ma essi avranno la forma di byte e non di linguaggio assemblatore.

Assembler

L'assembler accetta tutti i codici mnemonici standard nel formato standard, con l'eccezione che le virgole vanno sostituite da punti. I principali, comandi disponibili per l'assembler sono:

1 — assemblaggio in memoria: il file immesso per mezzo del file editor viene tradotto in codice macchina e piazzato in memoria. Si possono unire programmi a programmi già assemblati caricando programmi in linguaggio macchina in memoria (col file editor) ed iniziando poi l'assemblaggio del secondo programma al byte successivo al termine del primo, superando così ogni problema derivante dalla limita-

zione di un file a 255 linee. Notate che variabili ed etichette del primo programma vanno ridichiarate per il secondo.

2 – assemblaggio senza introduzione in memoria: il file viene assemblato con una lista completa degli indirizzi e del loro contenuto, ma la memoria resta inalterata.

3 – listing dei soli errori: vengono stampate le sole istruzioni contenenti errori, con un'indicazione del tipo di errore.

4 – listing completo: si stampa il listing completo comprese le indicazioni di errore. Notate che se ci sono due errori sulla stessa linea, solo uno verrà indicato. Assemblaggi successivi segnaleranno gli errori rimanenti una volta corretto il primo gruppo.

L'assembler fornisce sette 'direttive' che non compaiono nel programma in linguaggio macchina, ma modificano il modo di assemblare:

1 – ORG indirizzo: Questa direttiva indica che l'istruzione successiva in linguaggio assemblatore dev'essere assemblata all'indirizzo specificato – le istruzioni seguenti proseguiranno da quell'indirizzo. Un singolo programma in linguaggio assemblatore può contenere parecchie direttive ORG indicanti sezioni di programma da piazzare in aree di memoria completamente diverse.

2 – PRT: Seguendo questa direttiva, l'output del programma assemblato passa dallo schermo alla stampante.

3 – SYM: Indica che la tavola simboli, che contiene i valori delle variabili e delle etichette, dev'essere seguente il listing.

4 – END: Questa direttiva termina l'assemblaggio – non deve essere per forza al termina di un programma. Quando è usata come ultima linea di un programma, il suo indirizzo segnala il primo byte di memoria libero che segue il programma assemblato.

5 – BYT: Permette di specificare su una sola linea una serie di valori da un byte, separati da punti. I valori verranno immessi direttamente in memoria.

6 – DBY: Simile a BYT, eccetto che il valore specificato può essere nell'intervallo dei due byte (0-65535). I due byte verranno posti in memoria col byte alto per primo. WRD è identica a DBY eccetto che viene messo per primo il byte basso. Notate che i programmi assemblati possono essere salvati come file di codice macchina via Monitor, una volta posti in memoria.

APPENDICE C
MASTERCODE: TABELLA DELLE VARIABILI

AD	Indirizzo corrente in memoria;
AM	(Assemblaggio in memoria) flag usato nell'assembler;
BASE	Base numerica corrente per le conversioni;
CO	COntinua nel monitor / COmando nel file editor;
DEV	Indica il dispositivo per caricamento/salvataggio;
ES	Usata nel file editor per registrare le linee vuote;
EA	(End Address=indirizzo finale) usata nel monitor;
EC	(Error Count=contatore di errori) nell'assemblaggio;
EN	(Error Number=numero dell'errore) usata nell'assemblaggio per indicare il tipo di errore;
EO	(Error Only listing=listing dei soli errori) flag usato nell'assembler;
ERR	Usata per segnalare condizioni di errore;
ERR$	Messaggi di errore per l'assembler;
EXIT	Le si assegna un valore se l'assembler incontra la direttiva END;
FALSE	Valore logico (=0);
FI$	Matrice del file principale nel file editor;
FL	Linea cui terminare il listing o da cancellare nel file editor;
FM	Numero di righe in FI$;
FNDEC	Converte cifre decimali in esadecimali ASCII;
FNHEX	Converte cifre esadecimali in decimali;
FP	(Finish Pointer=puntatore al termine) usata da list e delete nel file editor;
H	Usata in routine di conversione – H$ convertita in decimale;
H$	Stringa generica di ingresso e uscita per numeri esadecimali;
IN$	Variabile generica usata per l'input;
LN	(Line Number=numero di linea) usata nel file editor;
PTR	Puntatore usato nell'esame delle istruzioni in linguaggio assemblatore;
PTR$	Contiene l'ordine degli elementi di FI$;
OP	Tipo dell'operando: assembler e disassembler;

O$	Stringa generica di uscita;
O1$	Stringa generica usata durante la visualizzazione su schermo del contenuto della memoria;
O2$	Come la precedente, usata anche nel disassembler;
O3$	Come O1$;
PASS	Passo corrente dell'assemblaggio in due passi;
PO	Puntatore al tipo di codice mnemonico;
Q	Variabile di ciclo usata nell'assembler;
Q1	Indirizzo iniziale della linea che viene assemblata;
Q3	Variabile di ciclo usata nell'assembler;
Q1$	Variabile temporanea usata nel formato d'uscita dell'assembler;
RESULT	Uscita del valutatore di espressioni;
SA	(Start Address=indirizzo iniziale) usata da parecchie routine;
SE	Numero corrente di simboli durante l'assemblaggio;
SL	(Start Line=linea iniziale) usata in list e delete nel file; editor;
SM	Massimo numero di simboli nella tavola simboli;
SP	Puntatore iniziale per list e delete nel file editor;
ST	Variabile di sistema in BASIC;
ST$	(Symbol Table=tavola simboli) usata in assembler;
SY	Usata per indicare il trasferimento della tavola simboli in assembler
T, TA, TB, T0, T1, T2, ecc.	Variabili numeriche temporanee usate in diversi moduli;
T$	Variabile stringa temporanea usata in diversi moduli;
TA$	Tabelle di decodifica per assembler/disassembler;
T1$	Variabile temporanea usata in diversi moduli;
TERM	Risultato temporaneo nel valutatore di espressioni;
TRUE	Valore logico (=−1);
X1	Variabile di ciclo usata nel caricatore esadecimale;
XY	Variabile di ciclo usata nel file editor;
XZ	Come sopra.

APPENDICE D

TABELLA DELLE SUBROUTINE DEL PROGRAMMA MASTERCODE

10000	Inizializzazione generale;
10100	Routine di controllo del monitor;
11000	Conversione decimale - esadecimale;
11100	Acquisizione di un byte dalla memoria;
11200	Ingresso dell'indirizzo finale;
11250	Ingresso del nome del file;
11850	Richiesta di continuazione;
11950	Conversione esadecimale - decimale;
12050	Ingresso dell'indirizzo iniziale;
12200	Caricamento di caratteri esadecimali in stringhe;
13000	Acquisizione di 1 byte dall'utente;
13100	Modifica della memoria;
13300	Trasferimento da memoria a schermo;
13500	Esecuzione del codice macchina;
14100	Salvataggio del codice macchina;
14300	Caricamento del codice macchina;
15300	Formato dell'operando;
15450	Formato dell'operando per l'indirizzamento in accumulatore;
15550	Formato dell'operando per l'indirizzamento implicito;
15550	Formato dell'operando per l'indirizzamento immediato;
15600	Formato dell'operando per l'indirizzamento relativo;
15700	Disassemblaggio di un'istruzione;
15800	Disassemblaggio di un'area di memoria;
19000	Inizializzazione tabelle di decodifica;
20000	Routine di controllo dell'assembler;
23020	Ricerca del numero di linea nel file;
23100	Aggiunta di una linea al file;
23300	Cancellazione di una linea dal file;

23400	Listing di una linea del file;
23500	Acquisizione dei puntatori di inizio e fine;
23600	Caricamento di un file da un dispositivo;
23700	Salvataggio di un file su un dispositivo;
23900	Rimozione degli spazi iniziali;
24000	Acquisizione del numero di linea dall'input della linea;
24200	Ingresso delle linee iniziale e finale;
24300	Inizializzazione del file;
24400	Listing delle linee;
24500	Cancellazione delle linee;
24600	Ingresso delle linee;
24700	Rinumerazione del file;
24800	Routine di controllo del file editor;
25000	Aggiunte da memoria al file;
25500	Cambiamento del numero di dispositivo;
26000	Ricerca di un simbolo fino ai due punti;
26100	Determinazione del tipo di operando usato;
26300	Valutazione del codice operativo;
26400	Routine di controllo del passo 1;
26500	Acquisizione della lunghezza dell'istruzione macchina;
26600	Calcolo della lunghezza della direttiva;
26900	Trasferimento della tavola simboli;
27000	Valutazione dell'operando;
27200	Valutazione della direttiva;
27400	Valutazione dell'operando immediato;
27500	Valutazione dell'operando relativo;
27600	Routine di controllo del passo 2;
28000	Routine assembler per gli errori;
28100	Stampa di IN$;
28150	Ricerca di un simbolo fino a non-cifra/non-lettera;
28250	Ricerca di un'etichetta in tavola simboli;
28300	Valutazione di un'etichetta o di un numero;
28500	Valutazione di un termine;
28600	Valutazione di un'espressione;
28700	Aggiunta di un simbolo alla tavola simboli;
28850	Test per i codici operativi mnemonici.

APPENDICE E
TABELLA DELLE ROUTINE ROM RICHIAMATE

A38A	Ricava il primo indirizzo di ritorno sulla pila;
A47A	Stampa 'READY' e torna in modo diretto;
A4A4	Inserisce una linea nel file BASIC;
A533	Ricollega il file BASIC;
A613	Converte il numero di linea contenuto in 14-15 esadecimali nell'indirizzo d'inizio linea in 5F-60 esadec.;
A65E	Esegue CLR;
A7F7	Converte il valore contenuto nell'accumulatore a virgola mobile #1 in un intero senza segno;
A831	Esegue END;
A8E0	Stampa 'RETURN WITHOUT GOSUB' e torna in modo diretto;
A8E3	Stampa 'UNDEFINED STATEMENT' e torna in modo diretto;
AD8A	Acquisisce un numero in virgola mobile dal programma BASIC e lo pone nell'accumulatore v.m. #1;
AEF1	Valuta un'espressione fra parentesi;
AEFD	Controlla che il carattere seguente nel programma BASIC sia una virgola, altrimenti stampa 'SYNTAX ERROR';
AEFF	Controlla che il carattere seguente nel programma BASIC sia uguale al contenuto dell'accumulatore, altrimenti stampa 'SYNTAX ERROR';
B248	Stampa 'ILLEGAL QUANTITY' e torna al modo diretto;
B7F7	Converte l'accumulatore a virgola mobile #1 in un intero senza segno;
B391	Converte un intero senza segno contenuto in 14-15 in un numero in virgola mobile contenuto nell'accumulatore v.m. #1;
B828	Fine della routine POKE;
B86A	Somma i numeri in virgola mobile degli accumulatori v.m. #1 e #2. Il risultato va nell'accumulatore #1.;

BA8C	Copia il numero in virgola mobile indicato dall'accumulatore (byte basso) e dal registro Y (byte alto) nell'accumulatore v.m. #2;
E15F	Esegue un salvataggio da memoria su dispositivo;
E175	Esegue caricamento o verifica da dispositivo;
E1D4	Acquisisce i parametri per load e save dal programma BASIC;
FFF0	Routine KERNAL che assegna o ricava la posizione del cursore.

APPENDICE F
TABELLA DEI CARATTERI DI CONTROLLO

COME RAPPRESENTATI NEL PROGRAMMA MASTERCODE

[CDW]	– Cursore giù
[CUP]	– Cursore su
[CLH]	– Pulizia dello schermo
[GRN]	– Control +6
[BLU]	– Control +7
[RON]	– Reverse On
[ROF]	– Reverse Off

Il libro apre nuovi orizzonti a tutti coloro che sono interessati alla programmazione in linguaggio macchina del Commodore 64.
La prima parte fornisce il listato completo e commentato del programma Mastercode, un potente assemblatore scritto in BASIC.
La seconda parte presenta una collezione di routine in codice macchina, che arricchiscono il BASIC standard della macchina con quattordici nuovi programmi.
Le descrizioni, ampie e precise, delle diverse routine costituiscono un'introduzione alle più importanti tecniche di programmazione in linguaggio macchina, danno preziosi suggerimenti per il miglior utilizzo della ROM del calcolatore.

L. 29.000 Cod. 572D 88-7056-175-5

170

Linguaggio Macchina del commodore 64

David Lawrence
Mark England

GRUPPO
EDITORIALE
JACKSON